Advanced Message Processing System (AMPS) User Guide

Advanced Message Processing System (AMPS) User Guide

5.0

Publication date Dec 27, 2016

Table of Contents

Part I. Introduction and Overview

Chapter 1. Introduction to 60East Technologies AMPS

Thank you for choosing the Advanced Message Processing System (AMPS™) from 60East Technologies®. AMPS is a feature-rich message processing system that delivers previously unattainable low-latency and high-throughput performance to users. AMPS provides both publish-and-subscribe messaging and high-performance message queueing.

1.1. Product Overview

AMPS, the Advanced Message Processing System, is built around an incredibly fast messaging engine that supports both publish-subscribe messaging and queuing. AMPS combines the capabilities necessary for scalable high-throughput, low-latency messaging in realtime deployments such as in financial services. AMPS goes beyond basic messaging to include advanced features such as high availability, historical replay, aggregation and analytics, content filtering and continuous query, last value caching, focus tracking, and more.

Furthermore, AMPS is designed and engineered specifically for next generation computing environments. The architecture, design and implementation of AMPS allows the exploitation of parallelism inherent in emerging multi-socket, multi-core commodity systems and the low-latency, high-bandwidth of 10Gb Ethernet and faster networks. AMPS is designed to detect and take advantage of the capabilities of the hardware of the system on which it runs.

AMPS does more than just route and deliver messages. AMPS was designed to lower the latency in real-world messaging deployments by focusing on the entire lifetime of a message from the message's origin to the time at which a subscriber takes action on the message. AMPS considers the full message lifetime, rather than just the "in flight" time, and allows you to optimize your applications to conserve network bandwidth and subscriber CPU utilization -- typically the first elements of a system to reach the saturation point in real messaging systems.

AMPS offers both topic and content based subscription semantics, which makes it different than most other messaging platforms. Some of the highlights of AMPS include:

- Topic and content based publish and subscribe
- Message queuing, including content-based filtering and configurable strategies for delivery fairness
- Client development kits for popular programming languages such as Java, C#, C++, C and Python
- Built in support for FIX, NVFIX, JSON, BSON, BFlat, Google Protocol Buffer and XML messages. AMPS also supports uninterpreted binary messages, and allows you to create composite message types from existing message types.
- State-of-the-World queries
- Historical State-of-the-World queries
- Easy to use command interface
- Full Perl-compatible regular expression matching
- Content filters with SQL92 **WHERE** clause semantics
- Built-in latency statistics and client status monitoring

- Advanced subscription management, including delta publish and subscriptions and out-of-focus notifications
- Basic CEP capabilities for real-time computation and analysis
- Aggregation within topics and joins between topics, including joins between different message types
- Replication for high availability
- Fully queryable transaction log
- Message replay functionality
- Fully-integrated authentication and entitlement system, including content-based entitlement for fine-grained control
- Optional encryption (SSL) between client and server. *In this release, SSL support is provided as a preview.*
- Extensibility API for adding message types, user-defined functions, user-specified actions, authentication, and entitlement functionality

1.2. Software Requirements

AMPS is supported on the following platforms:

- Linux 64-bit (2.6 kernel or later) on x86 compatible processors

> While 2.6 is the minimum kernel version supported, AMPS will select the most efficient mechanisms available to it and thus reaps greater benefit from more recent kernel and CPU versions.

1.3. Organization of this Manual

This manual is divided into the following parts:

- Part I presents introductory material and a brief overview of AMPS
- Part II explains the features of AMPS, including information on the following features:
  - Publish and Subscribe
  - Content Filtering
  - Transactional Messaging and Bookmark Subscriptions
  - Message Queues
  - Message Types
  - State of the World (SOW)

    State of the World topics enable many of the other advanced features in AMPS, such as:

    - Aggregating Data with View Topics
    - Conflated Topics
    - Delta Messaging
    - Out-of-Focus Messages (OOF)

    This section also contains detailed chapters on specific topics, such as the AMPS filter language. Both application developers and administrators should become familiar with this section.

- Part III discusses AMPS deployment and operations, including:
    - Running AMPS as a Linux Service
    - Logging
    - Event Topics
    - Monitoring Interface
    - Automating AMPS With Actions
    - Replication and High Availability
    - Operation and Deployment
    - Securing AMPS
    - Troubleshooting AMPS

    This section is most useful for those with a focus on AMPS operations, although the information presented here is helpful for developers who want to design high-performance, high-availability applications that are easy to deploy and maintain.

- Part IV presents information about using AMPS to build applications including:
    - Sample Use Cases

1.4. Document Conventions

This manual is an introduction to the 60East Technologies AMPS product. It assumes that you have a working knowledge of Linux, and uses the following conventions.

Table 1.1. Documentation Conventions

Construct	Usage
text	standard document text
`code`	inline code fragment
variable	variables within commands or configuration
	usage tip or extra information

Construct	Usage
⚠	usage warning
`required`	required parameters in parameter tables
optional	optional parameters in parameter tables

Additionally, here are the constructs used for displaying content filters, XML, code, command line, and script fragments.

```
(expr1 = 1) OR (expr2 = 2) OR (expr3 = 3) OR (expr4 = 4) OR (expr5 = 5) OR
(expr6 = 6) OR (expr7 = 7) OR (expr8 = 8)
```

Command lines will be formatted as in the following example:

```
find . -name *.java
```

1.5. Obtaining Support

For an outline of your specific support policies, please see your 60East Technologies License Agreement. Support contracts can be purchased through your 60East Technologies account representative.

Support Steps

You can save time if you complete the following steps before you contact 60East Technologies Support:

1. Check the documentation. The problem may already be solved and documented in the *User's Guide* or reference guide for the product. 60East Technologies also provides answers to frequently asked support questions on the support web site at http://support.crankuptheamps.com.

2. Isolate the problem.

   If you require Support Services, please isolate the problem to the smallest test case possible. Capture erroneous output into a text file along with the commands used to generate the errors.

3. Collect your information.

   - Your product version number.
   - Your operating system and its kernel version number.
   - The expected behavior, observed behavior and all input used to reproduce the problem.
   - Submit your request.
   - If you have a minidump file, be sure to include that in your email to `crash@crankuptheamps.com`.

The AMPS version number used when reporting your product version number follows a format listed below. The version number is composed of the following:

```
MAJOR.MINOR.MAINTENANCE.HOTFIX.TIMESTAMP.TAG
```

AMPS Versioning and Certification

Each AMPS version number component has the following breakdown:

Table 1.2. Version Number Components

Component	Description	Minimum Verification
`MAJOR`	Increments when there are any backward-incompatible changes in functionality, file formats, client network formats or configuration; or when deprecated functionality is removed. May introduce major new functionality or include internal improvements that introduce major behavioral changes.	Megacert
`MINOR`	Increments when functionality is added in a backwards-compatible way, or when functionality is deprecated. May include internal improvements, including internal improvements that introduce minor behavioral changes or changes to network formats used only by the AMPS server (such as replication).	Megacert
`MAINTENANCE`	Increments with standard bug fixing and maintenance. May introduce behavioral changes to fix incorrect behavior, to enhance performance, or to enable a feature to work as intended. May include internal enhancements that do not introduce behavioral changes.	Kilocert
`HOTFIX`	A release for a critical defect impacting a customer. A hotfix release is designed to be 100% compatible with the release it fixes (that is, a release with same MAJOR.MINOR.MAINTENANCE version). May introduce behavioral changes to fix incorrect behavior. May document previously undocumented features or extend surface area to improve usability for existing features.	Cert
`TIMESTAMP`	Proprietary build timestamp.	(does not affect verification level)
`TAG`	Identifier that corresponds to precise code used in the release.	(does not affect verification level)

The certification levels are defined in the following table. Notice that, in all cases, 60East will certify at a higher level if time permits or if a change involves a critical part of AMPS (such as replication or internal utility classes that are widely used).

Table 1.3. Certification Level Definitions

Certification Level	Description	Time to Certify
Megacert	Performance and long-haul testing. Full regression suite and stress-testing suite, including replication testing and application scenario tests.	less than 2 weeks

Certification Level	Description	Time to Certify
	Full unit testing suite, including new unit tests to verify correct behavior of bugfixes in this release.	
Kilocert	Full regression suite and stress-testing suite, including replication testing and application scenario tests. Full unit testing suite, including new unit tests to verify correct behavior of bugfixes in this release.	less than 1 week
Cert	Full unit testing suite, including new unit tests to verify correct behavior of bugfixes in this release. Replication testing suite if release affects replication code.	4 hours

Contacting 60East Technologies Support

Please contact 60East Technologies Support Services according to the terms of your 60East Technologies License Agreement.

Support is offered through the United States:

Toll-free:	(888) 206-1365
International:	(702) 979-1323
FAX:	(888) 216-8502
Web:	http://www.crankuptheamps.com
E-Mail:	`sales@crankuptheamps.com`
Support:	`support@crankuptheamps.com`

Chapter 2. Getting Started

Chapter 2 is for users who are new to AMPS and want to get up and running on a simple instance of AMPS. This chapter will walk new users through the file structure of an AMPS installation, configuring a simple AMPS instance and running the demonstration tools provided as part of the distribution to show how a simple publisher can send messages to AMPS.

2.1. Installing AMPS

To install AMPS, unpack the distribution for your platform where you want the binaries and libraries to be stored. For the remainder of this guide, the installation directory will be referred to as `$AMPSDIR` as if an environment variable with that name was set to the correct path.

Within `$AMPSDIR` the following sub-directories listed in Table 2.1.

Table 2.1. AMPS Distribution Directories

Directory	Description
bin	AMPS engine binaries and utilities
docs	Documentation
lib	Library dependencies
sdk	Include files for the AMPS extension API

AMPS client libraries are available as a separate download from the AMPS web site. See the AMPS developer page at http://www.crankuptheamps.com/developer to download the latest libraries.

2.2. Starting AMPS

The AMPS Engine binary is named `ampServer` and is found in `$AMPSDIR/bin`. Start the AMPS engine with a single command line argument that includes a valid path to an AMPS configuration file. You use the configuration file to enable and configure the AMPS features that your application will use. This guide discusses the most commonly-used configuration options for each feature, and the full set of options is described in the *AMPS Configuration Guide*.

The AMPS server can generate a sample configuration file with the `--sample-config` option. For example, you can save the sample configuration file to `$AMPSDIR/amps_config.xml` with the following command line:

```
$AMPSDIR/bin/ampServer --sample-config > $AMPSDIR/amps_config.xml
```

Once you have a configuration file saved to `$AMPSDIR/amps_config.xml` you can start AMPS with that file as follows:

```
$AMPSDIR/bin/ampServer $AMPSDIR/amps_config.xml
```

The sample configuration file generated by AMPS includes a very minimal configuration. The client evaluation kits include a sample configuration file that sets up AMPS to work with the samples, and the *AMPS Configuration Guide* contains a full description of the configuration items with sample configuration snippets.

> **Tip:** AMPS uses the current working directory for storing files (logs and persistence) for any relative paths specified in the configuration. While this is important for real deployments, the demo configuration used in this chapter does not persist anything, so you can safely start AMPS from any working directory using this configuration.

> **Tip:** On older processor architectures, `ampServer` will start the `ampServer-compat` binary. The `ampServer-compat` binary avoids using hardware instructions that are not available on these systems.
>
> You can also set the `AMPS_PLATFORM_COMPAT` environment variable to force `ampServer` to start the `ampServer-compat` binary. 60East recommends using this option only on systems that do not support the hardware instructions used in the standard binary. The `ampServer-compat` binary will not perform as well as `ampServer`, since it uses fewer hardware optimizations.

If your first start-up is successful, you should see AMPS display a simple message similar to the following to let you know that your instance has started correctly.

```
AMPS 5.0.X.X.973814.e1a57f7 - Copyright (c) 2006-2016 60East Technologies
 Inc.
(Built: 2015-02-15T00:26:45Z)
For all support questions: support@crankuptheamps.com
```

If you see this, congratulations! You have successfully cranked up the AMPS!

2.3. Admin View of the AMPS Server

When AMPS has been started correctly, you can get an indication as to whether AMPS is running or not by connecting to its admin port with a browser at `http://<host>:<port>` where *<host>* is the host the AMPS instance is running on and *<port>* is the administration port configured in the configuration file.

When successful, a hierarchy of information regarding the instance will be displayed. If you've started AMPS using the sample configuration file, try connecting to `http://localhost:8085/amps`. For more information on the monitoring capabilities, please see the *AMPS Monitoring Reference Guide*, available from the 60East documentation site at `http://docs.crankuptheamps.com/`.

2.4. Interacting with AMPS Using Spark

AMPS provides the `spark` utility as a command line interface to interacting with an AMPS server. `spark` provides many of the capabilities of the AMPS client libraries through this interface. The utility lets you execute commands like `'subscribe'`, `'publish'`, `'sow'`, `'sow_and_subscribe'` and `'sow_delete'`.

You can read more about `spark` in the `spark` chapter of the AMPS User Guide. Other useful tools for troubleshooting AMPS are described in the *AMPS Utilities Guide*.

2.5. Next Steps

The next step is to configure your own instance of AMPS to meet your messaging needs. The AMPS configuration is covered in more detail in *AMPS Configuration Reference Guide*

After you have successfully configured your own instance, there are two paths where you can go next.

One path is to continue using this guide and learn how to configure, administer and customize AMPS in depth so that it may meet the needs of your deployment. If you are a system administrator who is responsible for the deployment, availability and management of data to other users, then you may want to focus on this User Guide first.

The other path introduces the AMPS Client APIs. This path is targeted at software developers looking to integrate AMPS into their own solutions. 60East provides client libraries for C, C++, C#, Java and Python. These libraries are available for download from the 60East website. The website also includes evaluation kits designed to help programmers quickly get started with AMPS. For developers, the basic functionality of the AMPS server is explained in this User Guide. The Developer Guides and API documentation explain how to use that particular client library to create applications that use AMPS functionality.

Chapter 3. Spark

AMPS contains a command-line client, `spark`, which can be used to run queries, place subscriptions, and publish data. While it can be used for each of these purposes, `spark` is provided as a useful tool for informal testing and troubleshooting of AMPS instances. For example, you can use `spark` to test whether an AMPS instance is reachable from a particular system, or use `spark` to perform *ad hoc* queries to inspect the data in AMPS.

This chapter describes the commands available in the `spark`. For more information on the features available in AMPS, see the relevant chapters in the *AMPS User Guide*.

The `spark` utility is included in the `bin` directory of the AMPS install location. The `spark` client is written in Java, so running `spark` requires a Java Virtual Machine for Java 1.6 or later.

To run this client, simply type `./bin/spark` at the command line from the AMPS installation directory. AMPS will output the help screen as shown below, with a brief description of the `spark` client features.

```
%> ./bin/spark
===============================
- Spark - AMPS client utility -
===============================
Usage:

    spark help [command]

Supported Commands:

    help
    ping
    publish
    sow
    sow_and_subscribe
    sow_delete
    subscribe

Example:

    %> ./spark help sow

Returns the help and usage information for the 'sow' command.
```

Example 3.1. Spark Usage Screen

3.1. Getting help with spark

Spark requires that a supported command is passed as an argument. Within each supported command, there are additional unique requirements and options available to change the behavior of Spark and how it interacts with the AMPS engine.

For example, if more information was needed to run a `publish` command in Spark, the following would display the help screen for the Spark client's `publish` feature.

```
%>./spark help publish
================================
- Spark - AMPS client utility -
================================
Usage:

  spark publish [options]

Required Parameters:

  server    -- AMPS server to connect to
  topic     -- topic to publish to

Options:

  authenticator -- Custom AMPS authenticator factory to use
  delimiter     -- decimal value of message separator character
                   (default 10)
  delta         -- use delta publish
  file          -- file to publish records from, standard in when omitted
  proto         -- protocol to use (amps, fix, nvfix, xml)
             (type, prot are synonyms for backward compatibility)
                   (default: amps)
  rate          -- decimal value used to send messages
             at a fixed rate.  '.25' implies 1 message every
                   4 seconds. '1000' implies 1000 messages per second.

Example:

  % ./spark publish -server localhost:9003 -topic Trades -file data.fix

    Connects to the AMPS instance listening on port 9003 and publishes
 records
    found in the 'data.fix' file to topic 'Trades'.
```

Example 3.2. Usage of spark publish Command

3.2. Spark Commands

Below, the commands supported by `spark` will be shown, along with some examples of how to use the various commands and descriptions of the most commonly-used options. For the full range of options provided by `spark`, including options provided for compatibility with previous `spark` releases, use the `spark help` command as described above.

publish

The `publish` command is used to publish data to a topic on an AMPS server.

Common Options - spark publish

Table 3.1. Spark publish options

Option	Definition
`server`	AMPS server to connect to.
`topic`	Topic to publish to.
delimiter	Decimal value of message separator character (default 10).
delta	Use delta publish (sends a `delta_publish` command to AMPS).
file	File to publish messages from, stdin when omitted. `spark` interprets each line in the input as a message. The file provided to this argument can be either uncompressed or compressed in ZIP format.
proto	Protocol type to use. In this release, `spark` supports `amps`, `fix`, `nvfix`, `json` and `xml`. Defaults to `amps`.
rate	Messages to publish per second. This is a decimal value, so values less than 1 can be provided to create a delay of more than a second between messages. '.25' implies 1 message every 4 seconds. '1000' implies 1000 messages per second.
type	For protocols and transports that accept multiple message types on a given transport, specifies the message type to use.

Examples

The examples in this guide will demonstrate how to publish records to AMPS using the `spark` client in one of the three following ways: a single record, a python script or by file.

```
%> echo '{ "id" : 1, "data": "hello, world!" }' |  \
   ./spark publish -server localhost:9007 -type json -topic order

   total messages published: 1 (50.00/s)
```

Example 3.3. Publishing a single XML message.

In Example 3.3 a single record is published to AMPS using the `echo` command. If you are comfortable with creating records by hand this is a simple and effective way to test publishing in AMPS.

In the example, the JSON message is published to the topic *order* on the AMPS instance. This publish can be followed with a `sow` command in `spark` to test if the record was indeed published to the *order*topic.

```
%> python -c "for n in xrange(100): print '{\"id\":%d}' % n" | \
   ./spark publish -topic disorder -type json -rate 50 \
   -server localhost:9007

   total messages published: 100 (50.00/s)
```

Example 3.4. Publishing multiple messages using python.

In Example 3.4 the `-c` flag is used to pass in a simple loop and print command to the python interpreter and have it print the results to `stdout`.

The python script generates 100 JSON messages of the form `{"id":0}`, `{"id":1}` ... `{"id":99}`. The output of this command is then *piped* to spark using the | character, which will publish the messages to the *disorder* topic inside the AMPS instance.

```
%> ./spark publish -server localhost:9007 -type json -topic chaos \
   -file data.json

   total messages published: 50 (12000.00/s)
```

Example 3.5. Spark publish from a file

Generating a file of test data is a common way to test AMPS functionality. Example 3.5 demonstrates how to publish a file of data to the topic *chaos* in an AMPS server. As mentioned above, `spark` interprets each line of the file as a distinct message.

sow

The `sow` command allows a `spark` client to query the latest messages which have been persisted to a topic. The SOW in AMPS acts as a database last update cache, and the `sow` command in `spark` is one of the ways to query the database. This `sow` command supports regular expression topic matching and content filtering, which allow a query to be very specific when looking for data.

For the `sow` command to succeed, the topic queried must provide a SOW. This includes SOW topics and views, queues, and conflated topics. These features of AMPS are discussed in more detail in the *User Guide*.

Common Options - spark sow

Table 3.2. Spark sow options

Option	Definition
`server`	AMPS server to connect to.
`topic`	Topic to query.
batchsize	Batch Size to use during query. A batch size > 1 can help improve performance, as described in the chapter of the *User Guide* discussing the SOW.
filter	The content filter to use.
proto	Protocol type to use. In this release, `spark` supports `amps`, `fix`, `nvfix`, `json` and `xml`. Defaults to `amps`.
orderby	An expression that AMPS will use to order the results.
topn	Request AMPS to limit the query response to the first N records returned.
type	For protocols and transports that accept multiple message types on a given transport, specifies the message type to use.

Examples

```
%> ./spark sow -server localhost:9007 -type json -topic order \
```

```
   -filter "/id = '1'"

{ "id" : 1, "data" : "hello, world" }
Total messages received: 1 (Infinity/s)
```

Example 3.6. spark SOW query

This `sow` command will query the *order* topic and filter results which match the xpath expression `/id = '1'`. This query will return the result published in Example 3.3.

If the topic does not provide a SOW, the command returns an error indicating that the command is not valid for that topic.

subscribe

The `subscribe` command allows a `spark` client to query all incoming messages to a topic in real time. Similar to the `sow` command, the `subscribe` command supports regular expression topic matching and content filtering, which allow a query to be very specific when looking for data as it is published to AMPS. Unlike the `sow` command, a subscription can be placed on a topic which does not have a persistent SOW cache configured. This allows a subscribe command to be very flexible in the messages it can be configured to receive.

Common Options - spark subscribe

Table 3.3. Spark subscribe options

Option	Definition
`server`	AMPS server to connect to.
`topic`	Topic to subscribe to.
delta	Use delta subscription (sends a `delta_subscribe` command to AMPS).
filter	Content filter to use.
proto	Protocol type to use. In this release, `spark` supports `amps`, `fix`, `nvfix`, `json` and `xml`. Defaults to `amps`.
ack	Enable acknowledgements when receiving from a queue. Notice that, when this option is provided, `spark` acknowledges messages from the queue, signalling to AMPS that the message has been fully processed. (See the *User Guide* chapter on AMPS message queues for more information.)
backlog	Request a `max_backlog` of greater than 1 when receiving from a queue. (See the *User Guide* chapter on AMPS message queues for more information.)
type	For protocols and transports that accept multiple message types on a given transport, specifies the message type to use.

Examples

```
 %> ./spark subscribe -server localhost:9007 -topic chaos \
                        -type json -filter "/name = 'cup'"

{ "name" : "cup", "place" : "cupboard" }
```

Example 3.7. Spark subscribe example

Example 3.7 places a subscription on the *chaos* topic with a filter that will only return results for messages where `/name = 'cup'`. If we place this subscription before the `publish` command in Example 3.5 is executed, then we will get the results listed above.

sow_and_subscribe

The `sow_and_subscribe` command is a combination of the `sow` command and the `subscribe` command. When a `sow_and_subscribe` is requested, AMPS will first return all messages which match the query and are stored in the SOW. Once this has completed, all messages which match the subscription query will then be sent to the client.

The `sow_and_subscribe` is a powerful tool to use when it is necessary to examine both the contents of the SOW, and the live subscription stream.

Common Options - spark sow_and_subscribe

Table 3.4. Spark sow_and_subscribe options

Option	Definition
`server`	AMPS server to connect to.
`topic`	Topic to query and subscribe to.
batchsize	Batch Size to use during query.
delta	Request delta for subscriptions (sends a `sow_and_delta_subscribe` command to AMPS)
filter	Content filter to use.
proto	Protocol type to use. In this release, `spark` supports `amps`, `fix`, `nvfix`, `json` and `xml`. Defaults to `amps`.
orderby	An expression that AMPS will use to order the SOW query results.
topn	Request AMPS to limit the SOW query results to the first N records returned.
type	For protocols and transports that accept multiple message types on a given transport, specifies the message type to use.

Examples

```
%> ./spark sow_and_subscribe -server localhost:9007 -type json \
                                -topic chaos -filter "/name = 'cup'"

{ "name" : "cup", "place" : "cupboard" }
```

Example 3.8. spark SOW and subscribe example

In Example 3.8 the same topic and filter are being used as in the `subscribe` example in Example 3.7. The results of this query initially are similar also, since only the messages which are stored in the SOW are returned. If a publisher were started that published data to the topic that matched the content filter, then those messages would then be printed out to the screen in the same manner as a `subscription`.

sow_delete

The `sow_delete` command is used to remove records from the SOW topic in AMPS. If a filter is specified, only messages which match the filter will be removed. If a file is provided, the command reads messages from the file and sends those messages to AMPS. AMPS will delete the matching messages from the SOW. If no filter or file is specified, the command reads messages from standard input (one per line) and sends those messages to AMPS for deletion.

It can be useful to test a filter by first using the desired filter in a `sow` command and make sure the recored returned match what is expected. If that is successful, then it is safe to use the filter for a `sow_delete`. Once records are deleted from the SOW, they are not recoverable.

Common Options - sow_delete

Table 3.5. Spark sow_delete options

Option	Definition
`server`	AMPS server to connect to.
`topic`	Topic to delete records from.
filter	Content filter to use. Notice that a filter of `1=1` is true for every message, and will delete the entire set of records in the SOW.
file	File from which to read messages to be deleted.
proto	Protocol type to use. In this release, `spark` supports `amps`, `fix`, `nvfix`, `json` and `xml`. Defaults to `amps`.
type	For protocols and transports that accept multiple message types on a given transport, specifies the message type to use.

Examples

```
%> ./spark sow_delete -server localhost:9007 \
   -topic order -type json -filter "/name = 'cup'"

   Deleted 1 records in 10ms.
```

Example 3.9. spark SOW delete example

With the `spark` command in Example 3.9, we are asking for AMPS to delete records in the topic *order* which match the filter `/name = 'cup'`. In this example, we delete the record we published and queried previously in the publish and sow spark examples, respectively. spark reports that one matching message was removed from the SOW topic.

ping

The spark ping command is used to connect to the amps instance and attempt to logon. This tool is useful to determine if an AMPS instance is running and responsive.

Common Options - spark ping

Table 3.6. Spark ping options

Option	Definition
server	AMPS server to connect to.
proto	Protocol type to use. In this release, spark supports amps, fix, nvfix, json and xml. Defaults to amps.

Examples

```
%> ./spark ping -server localhost:9007 -type json
Successfully connected to tcp://user@localhost:9007/amps/json
```

Example 3.10. Successful ping using spark

In Example 3.10, spark was able to successfully log onto the AMPS instance that was located on port 9007.

```
%> ./spark ping -server localhost:9119
Unable to connect to AMPS
(com.crankuptheamps.client.exception.ConnectionRefusedException: Unable to
connect to AMPS at localhost:9119).
```

Example 3.11. Unsuccessful ping using spark

In Example 3.11, spark was not able to successfully log onto the AMPS instance that was located on port 9119. The error shows the exception thrown by spark, which in this case was a ConnectionRefusedException from Java.

3.3. Spark Authentication

Spark includes a way to authenticate credentials for development. For example, to subscribe to a a specific user ID and password, simply provide them in the URI in the format user:password@host:port.

The command below shows how to use spark to subscribe to a server, providing the specified username and password to AMPS.

```
$AMPS_HOME/bin/spark subscribe -type json \
                               -server username:password@localhost:9007
```

AMPS provides the ability to implement custom authentication. Spark includes a default authentication scheme, which uses the username and password in the -server parameter as described above.

Authentication schemes for `spark` are implemented in Java, using the same interface that the AMPS Java client uses. To use a different authentication scheme with `spark`, you implement the `AuthenticatorFactory` interface in `spark` to return your custom authenticator, adjust the CLASSPATH to include the .jar file that contains the authenticator, and then provide the name of your `AuthenticatorFactory` on the command line. See the *AMPS Java Client* API documentation for details on implementing a custom `Authenticator`.

The command below explicitly loads the default factory, found in the `spark` package, without adjusting the CLASSPATH.

```
$AMPS_HOME/bin/spark subscribe -server username:password@localhost:9007 \
                               -type json -topic foo \
      -authenticator com.crankuptheamps.spark.DefaultAuthenticatorFactory
```

Part II. Understanding AMPS

Chapter 4. Publish and Subscribe

AMPS is a rich message delivery system. At the core of the system, the AMPS engine is highly-optimized for publish and subscribe delivery. In this style of messaging, publishers send messages to a message broker (such as AMPS) which then routes and delivers messages to the subscribers. "Pub/Sub" systems, as they are often called, are a key part of most enterprise message buses, where publishers broadcast messages without necessarily knowing all of the subscribers that will receive them. This decoupling of the publishers from the subscribers allows maximum flexibility when adding new data sources or consumers.

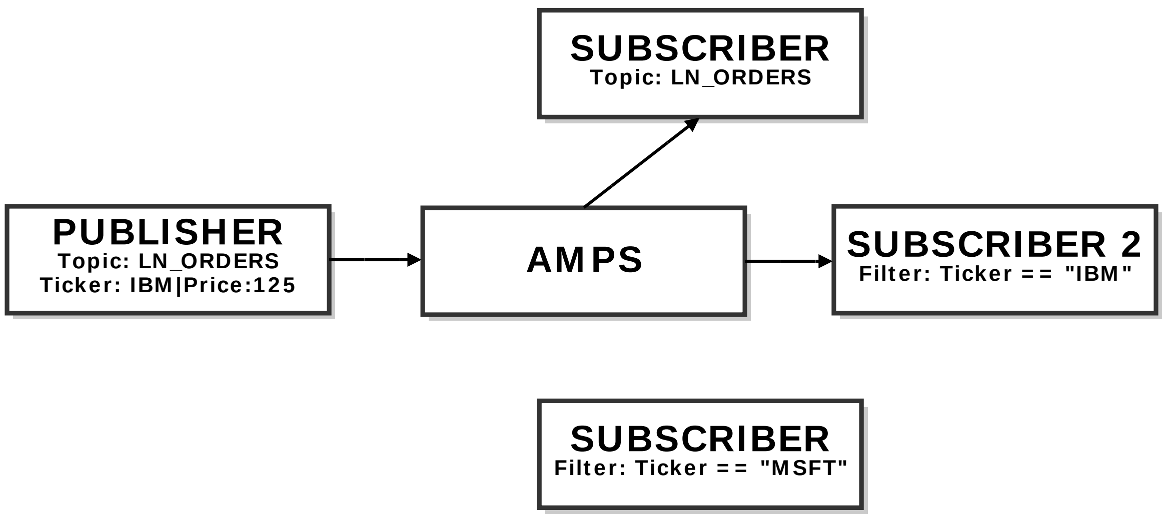


Figure 4.1. Publish and Subscribe

AMPS can route messages from publishers to subscribers using a topic identifier and/or content within the message's payload. For example, in Chapter 4, there is a Publisher sending AMPS a message pertaining to the `LN_ORDERS` topic. The message being sent contains information on Ticker "IBM" with a Price of 125, both of these properties are contained within the message payload itself (i.e., the message content). AMPS routes the message to Subscriber 1 because it is subscribing to all messages on the `LN_ORDERS` topic. Similarly, AMPS routes the message to Subscriber 2 because it is subscribed to any messages having the Ticker equal to "IBM". Subscriber 3 is looking for a different Ticker value and is not sent the message.

4.1. Topics

A topic is a string that is used to declare a subject of interest for purposes of routing messages between publishers and subscribers. Topic-based Publish and-Subscribe (e.g., Pub/Sub) is the simplest form of Pub/Sub filtering. All messages are published with a topic designation to the AMPS engine, and subscribers will receive messages for topics to which they have subscribed.

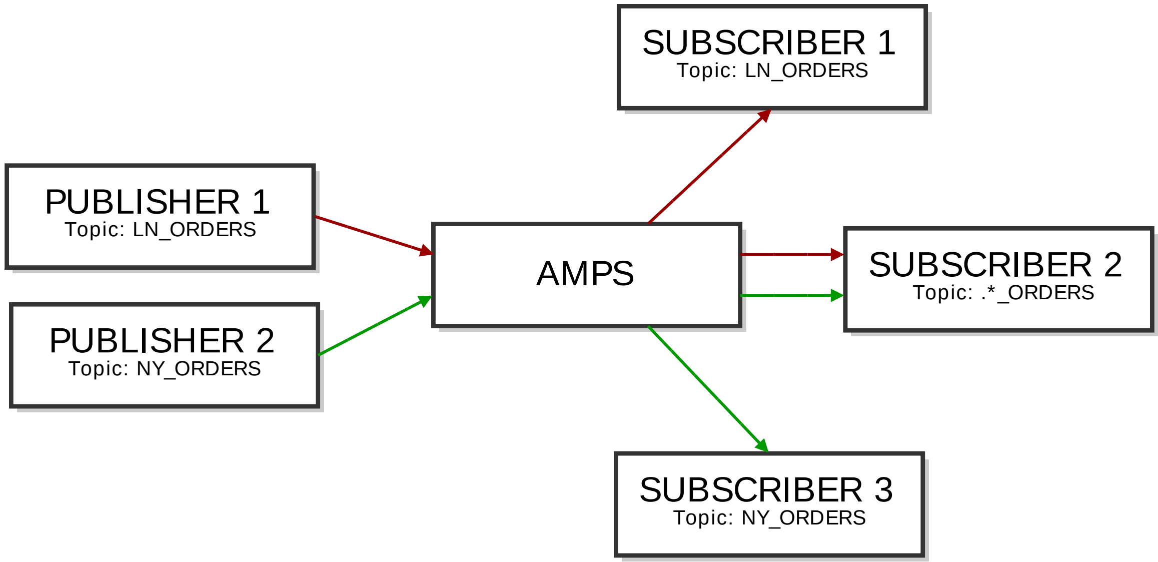


Figure 4.2. Topic Based Pub/Sub

For example, in Section 4.1, there are two publishers: Publisher 1 and Publisher 2 which publish to the topics `LN_ORDERS` and `NY_ORDERS`, respectively. Messages published to AMPS are filtered and routed to the subscribers of a respective topic. For example, Subscriber 1, which is subscribed to all messages for the `LN_ORDERS` topic will receive everything published by Publisher 1. Subscriber 2, which is subscribed to the regular expression topic `".*_ORDERS"` will receive all orders published by Publisher 1 and 2.

Regular expression matching makes it easy to create topic paths in AMPS. Some messaging systems require a specific delimiter for paths. AMPS allows you the flexibility to use any delimiter. However, 60East recommends using characters that do not have significance in regular expressions, such as forward slashes. For example, rather than using `northamerica.orders` as a path, use `northamerica/orders`.

AMPS does not restrict the characters that can be present in a topic name. However, notice that topic names that contain regular expression characters (such as `.` or `*`) will be interpreted as regular expressions by default, which may cause unexpected behavior.

Topics that begin with `/AMPS` are reserved. The AMPS server publishes messages to topics that begin with `/AMPS` as described in Chapter 21. Some versions of the AMPS client libraries may internally publish to `/AMPS/devnull`. Your applications should not publish to topics that begin with `/AMPS`, and publishes to those topics may fail.

Regular Expressions

With AMPS, a subscriber can use a regular expression to simultaneously subscribe to multiple topics that match the given pattern. This feature can be used to effectively subscribe to topics without knowing the topic names in advance. Note that the messages themselves have no notion of a topic pattern. The topic for a given message is unambiguously specified using a literal string. From the publisher's point of view, it is publishing a message to a topic; it is never publishing to a topic pattern.

Subscription topics are interpreted as regular expressions if they include special regular expression characters. Otherwise, they must be an exact match. Some examples of regular expressions within topics are included in Table 4.1.

Table 4.1. Topic Regular Expression Examples

Topic	Behavior
`^trade$`	matches only “trade”.
`^client.*`	matches “client”, “clients”, “client001”, etc.
`.*trade.*`	matches “NYSEtrades”, “ICEtrade”, etc.

For more information regarding the regular expression syntax supported within AMPS, please see the *Regular Expression* chapter in the *AMPS User Guide*.

AMPS can be configured to disallow regular expression topic matching for subscriptions. See the *AMPS Configuration Guide* for details.

4.2. Filtering Subscriptions By Content

One thing that differentiates AMPS from classic messaging systems is its ability to route messages based on message content. Instead of a publisher declaring metadata describing the message for downstream consumers, the publisher can simply publish the message content to AMPS and let AMPS examine the native message content to determine how best to deliver the message.

The ability to use content filters greatly reduces the problem of oversubscription that occurs when topics are the only facility for subscribing to message content. The topic space can be kept simple and content filters used to deliver only the desired messages. The topic space can reflect broad categories of messages and does not have to be polluted with metadata that is usually found in the content of the message. In addition, many of the advanced features of AMPS such as out-of-focus messaging, aggregation, views, and SOW topics rely on the ability to filter content.

Content-based messaging is somewhat analogous to database queries that include a `WHERE` clause. Topics can be considered tables into which rows are inserted (or updated). A subscription is similar to issuing a `SELECT` from the topic table with a `WHERE` clause to limit the rows which are returned. Topic-based messaging is analogous to a `SELECT` on a table with no limiting `WHERE` clause.

AMPS uses a combination of XPath-based identifiers and SQL-92 operators for content filtering. Some examples are shown below:

Example Filter for a JSON message

```
(/Order/Instrument/Symbol == 'IBM') AND
(/Order/Px >= 90.00 AND /Order/Px < 91.00)
```

Example Filter for an XML Message:

```

(/FIXML/Order/Instrmt/@Sym == 'IBM') AND (/FIXML/Order/@Px
       >= 90.00 AND /FIXML/Order/@Px < 91.0)
```

Example Filter for a FIX Message:

```
/35 < 10 AND /34 == /9
```

For more information about how content is handled within AMPS, check out the *Content Filtering* chapter in the *AMPS User Guide*.

Unlike some other messaging systems, AMPS lets you use a relatively small set of topics to categorize messages at a high level and use content filters to retrieve specific data published to those topics. Examples of good, broad topic choices:

`trades, positions, MarketData, Europe, alerts`

This approach makes it easier to administer AMPS, easier for publishers to decide which topics to publish to, and easier for subscribers to be sure that they've subscribed to all relevant topics.

4.3. Replacing Subscriptions

AMPS provides the ability to perform atomic subscription replacement. This allows you to replace the filter, change the topic, or update the options for a subscription.

The most common use for this capability is for an application to change the filter for a subscription. For example, a GUI that is providing a view of a set of orders may need to add or remove an order from the set of orders being displayed. By replacing the content filter with a filter that tracks the updated set of orders, the application can do this without missing messages, getting duplicate messages, or having to manage more than one subscription.

Replacing a filter is an atomic operation. That is, the application is guaranteed not to miss messages that are in both the original and replacement subscription, and is guaranteed to receive all messages for the new subscription as of the point at which the replacement happens.

When replacing a `sow_and_subscribe` command (described later in the guide), AMPS runs the SOW command again and provides any messages that were not previously in the result set to the application. See the section called "Replacing Subscriptions with SOW and Subscribe" for details.

Replacing the Content Filter on a Subscription

AMPS allows you to replace the content filter on an existing subscription. When this happens, AMPS begins sending messages on the subscription that match the new filter. When an application needs to bring more messages into scope, this can be more efficient than creating another subscription.

For example, an application might start off with a filter such as the following

```
/region = 'WesternUS'
```

The application might then need to bring other regions into scope, for example:

```
/region IN ('WesternUS', 'Alaska', 'Hawaii')
```

Replacing the Topic on a Subscription

AMPS allows a subscription to replace the topic on a subscription. When the topic is replaced, AMPS re-evaluates the subscription as it does when a filter is replaced. If the subscription is updated to include a topic that the user does not have permission to subscribe to, the `replace` operation succeeds, but no messages will be delivered on that topic.

Replacing the Options on a Subscription

AMPS allows a subscription to replace some of the options on the subscription. In this case, the subscription is evaluated as though the topic or filter has been replaced. Any new messages generated after the point of the subscription being replaced use the new options. However, AMPS does not replay or requery previous messages to apply the options. For example, if a `sow_and_subscribe` command did not previously specify Out-of-Focus tracking and adds this option, AMPS generates the appropriate Out-of-Focus messages from the replace point forward. AMPS does not recreate Out-of-Focus messages that would have previously been generated by the subscription.

4.4. Messages in AMPS

Communication between applications and the AMPS server uses AMPS messages. AMPS Messages are received or sent for every operation in AMPS. Each AMPS message has a specific type, and consists of a set of headers and a payload. The headers are defined by AMPS and formatted according to the protocol specified for the connection. Typically, applications use the standard `amps` protocol which uses a JSON document for headers. The payload, if one is present, is the content of the message, and is in the format specified by the message type.

Messages received from AMPS have the same format as messages to AMPS. These messages also have a specific type, with a header formatted according to the protocol and a payload of the specified message type. For example, AMPS uses `ack` messages, short for acknowledgement, to report the status of commands. AMPS uses `publish` messages to deliver messages on a subscription, and so on for other commands and other messages.

For example, when a client subscribes to a topic in AMPS, the client sends a `subscribe` message to AMPS that contains the information about the requested subscription and, by default, a request for an acknowledgement that the subscription has been processed. AMPS returns an `ack` message when the subscription is processed that indicates whether the subscription succeeded or failed, and then begins providing `publish` messages for new messages on the subscription.

Messages to and from AMPS are described in more detail in the *AMPS Command Reference*, available on the 60East website and included in the AMPS client SDKs.

Introduction to AMPS Headers

The *AMPS Command Reference* contains a full list of headers for each command. The table below lists some commonly-used headers.

Table 4.2. Basic AMPS Headers

Header	Description
Topic	The topic that the message applies to. For commands to AMPS, this is the topic that AMPS will apply the com-

Header	Description
	mand to. For messages from AMPS, this is the topic from which the message originated.
Command	The command type of message. Each message has a specific command type. For example, messages that contain data from a query over a SOW topic have a command of `sow`, while messages that contain data from a publish command have a command of `publish`, and messages that acknowledge a command to AMPS have a command type of `ack`.
CommandId	An identifier used to correlate responses from AMPS with an initial command. For example, `ack` messages returned by AMPS contain the CommandId provided with the command they acknowledge, and subscriptions can be updated or removed using the CommandId provided with the `subscribe` command.
SowKey	For messages received from a State of the World (or *SOW*) topic, an identifier that AMPS assigns to the record for this message. SOW topics are described in Chapter 7. This header is included on messages from a SOW topic by default. AMPS will omit this header when the subscription or SOW query includes the `no_sowkeys` option.
CorrelationId	A user-specified identifier for the message. Publishers can set this identifier on messages. AMPS does not parse, change, or interpret this identifier in any way. This header is limited to characters used in Base64 encoding.
Status	Set on `ack` messages to indicate the results of the command, such as `Success` or `Failure`.
Reason	Set on `ack` messages to indicate the reason for the Status acknowledgement.
Timestamp	Optionally set on `publish` messages and `sow` messages to indicate the time at which AMPS processed the message. To receive a timestamp, the SOW query or subscription must include the `timestamp` option on the command that creates the subscription or runs the query. The timestamp is returned in ISO-8601 format.

This section presents a few of the commonly-used headers. See the *AMPS Command Reference* for a full description of AMPS messages.

AMPS does not provide the ability to add custom header fields. However, AMPS composite message types provide an easy way to add an additional section to a message type that contains metadata for the message. Because composite message type parts fully support AMPS content filtering, this approach provides more flexibility and allows for more sophisticated metadata than simply adding a header field. See Section 17.3 for details.

4.5. Message Ordering

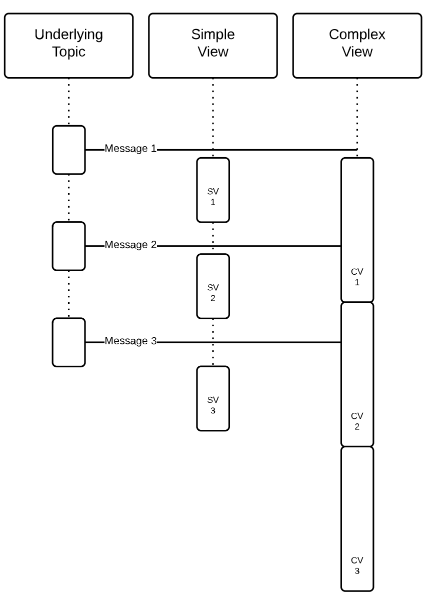
AMPS guarantees that, for each AMPS instance, all subscribers to a topic receive messages in the order in which AMPS received the messages (with the exception of messages that have been returned to a message queue for redelivery). Before a given message is delivered to a subscriber, all previous messages for that topic are delivered to the subscriber. AMPS does this by enforcing a total order across the instance for all messages received from publishers, including messages received via replication. When AMPS is using a transaction log, that order is preserved in the transaction log for the instance, and persists across instance restarts.

This guarantee also applies across topics for subscriptions that involve multiple topics, for all topics *except* views, queues, and conflated topics. Views and queues guarantee that every message on the view or the queue appears in the order in which the message was published. However, the computation involved in producing messages for views and queues may introduce some amount of processing latency, and AMPS does not delay messages on other topics while performing these computations. For a queue that provides `at-least-once` delivery, if a processor fails and returns a message to the queue, that message will be redelivered (which means that the new processor may receive the message out of order). Likewise, since a conflated topic is designed to conflate updates, AMPS does not attempt to preserve any ordering guarantees between a conflated topic and other topics in the instance.

Applications often use this guarantee to publish checkpoint messages, indicating some external state of the system, to a checkpoint topic. For example, you might publish messages marking the beginning of a business day to a checkpoint topic, `MARKERS`, while the `ORDERS` topic records the orders during that day. Subscribers to the regular expression `^(ORDERS|MARKERS)$` are guaranteed to receive the message that marks the business day before any of the messages published to the ORDERS topic for that day, since AMPS preserves the original order of the messages.

For messages constructed by AMPS, such as the output of a view, AMPS processes messages for each topic in the order in which they arrive, and delivers each calculated message to subscribers as soon as the calculation is finished and a message is produced. This keeps the latency low for each individual topic. However, this means that while AMPS guarantees the order in which messages are produced within each view, messages produced for views that do simple operations will generally take less time to be produced than messages for views that perform complex calculations or require more complicated serialization. This means that AMPS guarantees ordering within view topics, but does not guarantee that messages for separate view topics arrive in a particular order.

The figure below shows a possible ordering for messages received on an underlying topic and two views that use the topic:



Notice that within each topic, AMPS enforces an absolute order. However, the Simple View produces the results of Message 3 before the Complex View produces the results of Message 2.

Replicated Message Ordering

When providing messages received via *replication* (see Section 25.1), the principles on message ordering provided above still apply. AMPS records messages into the local transaction log in the order in which messages are received by the instance, and provides messages to subscribers in that order. AMPS uses the sequence of publishes assigned by the original publisher and the order assigned by the upstream instance to ensure that all replicated messages are received and recorded in order with no gaps or duplicates. AMPS does not enforce a global total ordering across a replication topology. This peer-to-peer approach means that an AMPS instance can continue accepting messages from publishers and providing messages to subscribers even when the remote side of a replication link is offline or if replication is delayed due to network congestion. However, if two messages are published to *different* instances at the same time by different publishers, the two instances may record a different *overall* message order for those messages, even though message order *from each publisher* is preserved.

Chapter 5. Content Filtering

AMPS allows a subscriber to specify a content filter using syntax similar to that of SQL-92's **WHERE** clause. Content filters are used to provide a greater level of selectivity in messages sent over a subscription than provided by topic alone. When using a content filter, only the messages matching the requested topic and the content filter are delivered to the subscribing client.

5.1. Syntax

AMPS implements content filtering using *expressions* that combine SQL-92 and XPath syntax. Instead of table columns, XPath expressions are used to identify values within a message. The syntax of the filter expression is based on a subset of the SQL-92 search condition syntax.

Each expression compares two values. A value can be either an *identifier* that specifies a value in a message, a *literal* value, such as `42` or `'IBM'`, or a regular expression as described in Chapter 6. Comparison is done with either a logical operator or an arithmetic operator.

For example, the following expression compares the `OrderQty` value in each message with a number:

```
/OrderQty > 42
```

The following expression compares two fields in the message:

```
/21694 < /14
```

A content filter is made up of one or more expressions, joined together by logical operators and grouped using parentheses. For example:

```
(expression1 OR expression2 AND expression3) OR (expression4 AND
            NOT expression5) ...
```

A content filter is evaluated left to right in precedence order. So in this example, `expression2` will be evaluated followed by `expression3` (since `AND` has higher precedence than `OR`), and if they evaluate to false, then `expression1` will be evaluated and so on.

Identifiers

AMPS identifiers use a subset of XPath to specify values in a message. AMPS identifiers specify the value of an attribute or element in an XML message, and the value of a field in a JSON, FIX or NVFIX message. Because the *identifier* syntax is only used to specify values, the subset of XPath does not need to include relative paths, arrays, predicates, or functions.

For example, when messages are in this XML format:

```
<Order update="full">
    <ClientID>12345</ClientID>
    <Symbol>IBM</Symbol>
    <OrderQty>1000</OrderQty>
</Order>
```

The following identifier specifies the `Symbol` element of an `Order` message:

```
/Order/Symbol
```

The following identifier specifies the `update` attribute of an `Order` message:

```
/Order/@update
```

For FIX and NVIX, you specify fields using `/` and the tag name. AMPS interprets FIX and NVFIX messages as though they were an XML fragment with no root element. For example, to specify the value of FIX tag `55` (symbol), use the following identifier:

```
/55
```

Likewise, for JSON or other types that represent an object, you navigate through the object structure using the `/` to indicate each level of nesting.

AMPS checks the syntax of identifiers, but does not try to predict whether an identifier will match messages within a particular topic. It is not an error to submit an identifier that can never match. For example, AMPS allows you to use an identifier like `/OrderQty` with a FIX topic, even though FIX messages only use numeric tags.

AMPS only supports identifiers that are valid step names in XPath. For example, AMPS does not guarantee that it can process or filter on a field named `Fits&Starts`.

Literals

String literals are indicated with single or double quotes. For example:

```
/FIXML/Order/Instrmt/@Sym = 'IBM'
```

AMPS supports the following escape sequences within string literals:

Table 5.1. Escape Sequences

Escape Sequence	Definition
\a	Alert
\b	Backspace
\t	Horizontal tab
\n	Newline
\f	Form feed
\r	Carriage return
\xHH	Hexadecimal digit where H is (0..9,a..f,A..F)
\OOO	Octal Digit (0..7)

Additionally, any character which follows a backslash will be treated as a literal character.

Numeric literals are either integer values or floating-point values. Integer values are all numerals, with no decimal point, and can have a value in the same range as a 64-bit integer. For example:

```
42
149
-273
```

Floating-point literals are all numerals with a decimal point:

```
3.1415926535
98.6
-273.0
```

or, in scientific notation:

```
31.4e-1
6.022E23
2.998e8
```

Literals can also be the Boolean values `true` or `false`.

Logical Operators

The logical operators are `NOT`, `AND`, and `OR`, in order of precedence. These operators have the usual Boolean logic semantics.

```
/FIXML/Order/Instrmt/@Sym = 'IBM' OR /FIXML/Order/Instrmt/@Sym = 'MSFT'
```

As with other operators, you can use parentheses to group operators and affect the order of evaluation

```
(/orderType = 'rush' AND /customerType IN ('silver', 'gold') )
OR  /customerType = 'platinum'
```

Arithmetic Operators

AMPS supports the arithmetic operators `+`, `-`, `*`, `/`, `%`, and `MOD` in expressions. The result of arithmetic operators where one of the operands is `NULL` is undefined and evaluates to `NULL`. Examples of filter expressions using arithmetic operators:

```
/6 * /14 < 1000

/Order/@Qty * /Order/@Prc >= 1000000
```

Numeric values in AMPS are always *typed* as either integers or floating point values. AMPS uses the following rules for type promotion when evaluating expressions:

1. If any of the values in the expression is `NaN`, the result is `NaN`.
2. Otherwise, if any of the values in the expression is floating point, the result is floating point.
3. Otherwise, all of the values in the expression are integers, and the result is an integer.

Notice that, for division in particular, the results returned are affected by the type of the values. For example, the expression `1 / 5` evaluates to `0` (since the result interpreted as an integer), while the expression `1.0 / 5` evaluates to `0.2` (since the result is interpreted as a floating point value).

When using mathematical operators in conjunction with filters, be careful about the placement of the operator. Some operators are used in the XPath expression as well as for mathematical operation (for example, the '/' operator in division). Therefore, it is important to separate mathematical operators with white space, to prevent interpretation as an XPath expression.

Comparison Operators

The comparison operators can be loosely grouped into equality comparisons and range comparisons. The basic equality comparison operators, in precedence order, are `==`, `=`, `>`, `>=`, `<`, `<=`, `!=`, and `<>`. If these binary operators are applied to two operands of different types, AMPS attempts to convert strings to numbers. If conversion succeeds, AMPS uses the numeric values. If conversion fails, strings are always greater than numbers.

The following table shows some examples of how AMPS compares different types.

Table 5.2. Comparison Operator Examples

Expression	Result
1 < 2	TRUE
10 < '2'	FALSE, '2' can be converted to a number
'2.000' <> '2.0'	TRUE, both are strings
10 < 'Crank It Up'	TRUE, strings are greater than numbers

There are also set and range comparison operators. The BETWEEN operator can be used to check the range values.

The range used in the **BETWEEN** operator is inclusive of both operands, meaning the expression `/A BETWEEN 0 AND 100` is equivalent to `/A >= 0 AND /A <= 100`

For example:

```
/FIXML/Order/OrdQty/@Qty BETWEEN 0 AND 10000

/FIXML/Order/@Px NOT BETWEEN 90.0 AND 90.5

(/price * /qty) BETWEEN 0 AND 100000
```

The `IN` operator can be used to perform membership operations on sets of values:

```
/Trade/OwnerID NOT IN ('JMB', 'BLH', 'CJB')

/21964 IN (/14*5, /6*/14, 1000, 2000)

/customer IN ('Bob', 'Phil', 'Brent')
```

When evaluating against a set of values, the `IN` operator typically provides better performance than using a set of `OR` operators. That is, a filter written as `/firstName IN ('Joe', 'Kathleen', 'Frank', 'Cindy', 'Mortimer')` will typically perform better than an equivalent filter written as `/firstName = 'Joe' OR /firstName = 'Kathleen' OR /firstName = 'Frank' OR /firstName = 'Cindy' OR /firstName = 'Mortimer'`.

AMPS includes two kinds of string comparison operators. The `BEGINS WITH`, `ENDS WITH`, and `INSTR` operators do literal matching on the contents of a string.

`BEGINS WITH` and `ENDS WITH` test whether a field begins or ends with the literal string provided. The operators return `TRUE` or `FALSE`:

```
/Department BEGINS WITH ('Engineering')

/path NOT BEGINS WITH ('/public/dropbox')

/filename ENDS WITH ('txt')

/price NOT ENDS WITH ('99')
```

AMPS allows you to use set comparisons with `BEGINS WITH` and `ENDS WITH`. In this case, the filter matches if the string in the field `BEGINS WITH` or `ENDS WITH` any of the strings in the set:

```
/Department BEGINS WITH ('Engineering', 'Research', 'Technical')

/filename ENDS WITH ('gif', 'png', 'jpg')
```

The `INSTR` operator allows you to check to see if one string occurs within another string. For this operator, you provide two string values. If the second string occurs within the first string, INSTR returns the position at which the second string starts, or 0 if the second string does not occur within the first string. Notice that the first character of the string is 1 (not 0). For example, the expression below tests whether the string `critical` occurs within the `/eventLevels` field.

```
INSTR(/eventLevels, "critical") != 0
```

AMPS also provides a more general comparison operator, `LIKE`, that allows for regular expression matching on string values. A pattern is used for the right side of the `LIKE` operator. For more on regular expressions and the `LIKE` comparison operator, please see Chapter 6.

The `BEGINS WITH` and `ENDS WITH` operators are usually more efficient than equivalent `LIKE` expressions, particularly when used to compare multiple patterns.

Conditional Operators

AMPS contains support for a ternary conditional `IF` operator which allows for a Boolean condition to be evaluated to `true` or `false`, and will return one of the two parameters. The general format of the `IF` statement is

```
IF ( BOOLEAN_CONDITIONAL, VALUE_TRUE, VALUE_FALSE)
```

In this example, the `BOOLEAN_CONDITIONAL` will be evaluated, and if the result is true, the `VALUE_TRUE` value will be returned otherwise the `VALUE_FALSE` will be returned.

For example:

```
SUM( IF(( (/FIXML/Order/OrdQty/@Qty > 500) AND
          (/FIXML/Order/Instrmt/@Sym ='MSFT')), 1, 0 ))
```

The above example returns a count of the total number of orders that have been placed where the symbol is MSFT and the order contains a quantity more than 500.

The `IF` can also be used to evaluate results to determine if results are `NULL` or `NaN`. This is useful for calculating aggregates where some values may be `NULL` or `NaN`. The NULL and NaN values are discussed in more detail in the section called "NULL, NaN and IS NULL".

For example:

```
SUM(/FIXML/Order/Instrmt/@Qty * IF(
   /FIXML/Order/Instmt/@Price IS NOT NULL, 1, 0))
```

NULL, NaN and IS NULL

XPath expressions are considered to be `NULL` when they evaluate to an empty or nonexistent field reference. In numeric expressions where the operands or results are not a valid number, the XPath expression evaluates to `NaN` (not a number). The rules for applying the `AND` and `OR` operators against `NULL` and `NaN` values are outlined in Table 6.2 and Table 6.3.

Table 5.3. Logical AND with NULL/NaN Values

Operand1	Operand2	Result
TRUE	NULL	NULL
FALSE	NULL	FALSE
NULL	NULL	NULL

Table 5.4. Logical OR with NULL/NaN Values

Operand1	Operand2	Result
TRUE	NULL	TRUE
FALSE	NULL	NULL
NULL	NULL	NULL

The `NOT` operator applied to a `NULL` value is also `NULL`, or "Unknown." The only way to check for the existence of a `NULL` value reliably is to use the `IS NULL` predicate. There also exists an `IS NAN` predicate for checking that a value is `NaN` (not a number.)

> ⚠ To reliably check for existence of a `NULL` value, you must use the `IS NULL` predicate such as the filter: `/Order/Comment IS NULL`

Working With Arrays

AMPS supports filters that operate on arrays in messages. There are two simple principles behind how AMPS treats arrays.

Binary operators that yield `true` or `false` (for example, `=`, `<`, `LIKE`) are *array aware*, as is the `IN` operator. These operators work on arrays as a whole, and evaluate every element in the array. *Arithmetic operators*, and other *scalar* *operators*, are *not* array aware, and use the first element in the array. With these simple principles, you can predict how AMPS will evaluate an expression that uses an array. For any operator, an empty array evaluates to NULL.

Let's look at some examples. For the purposes of this section, we will consider the following JSON document:

```
{ "data"  : [1, 2, 3, "zebra", 5],
  "other" : [14, 34, 23, 5] }
```

While these arrays are presented using JSON format for simplicity, the sample principles apply to arrays in other message formats.

Here are some examples of ways to use an array in an AMPS filter:

1. Determining if any element in an array meets a criteria. To determine this, you provide the identifier for the array, and use a comparison operator.

Table 5.5. Array contains element

Filter	Evaluates as
`/data = 1`	`true`
`/data = 'zebra'`	`true`
`/data = 42`	`false`
`/data LIKE 'z'`	`true`
`/other > 30`	`true`
`/other > 50`	`false`

2. Determine whether a specific value is at a specific position. To determine this, use the subscript operator `[]` on the XPath identifier to specify the position, and use the equality operator to check the value at that position.

Table 5.6. Element at specific position

Filter	Evaluates as
`/data[0] = 1`	`true`
`/data[3] = "zebra"`	`true`
`/data[1] = 1`	`false`
`/other[1] LIKE '4'`	`true`

3. Determine whether any value in one array is present in another array.

Table 5.7. Identical elements

Filter	Evaluates as
`/data = /other`	`true`

4. Determine whether an array contains one of a set of values.

Table 5.8. Set of values in an array

Filter	Evaluates as
`3 IN (/data)`	`true`
`/data IN (1, 2, 3)`	`true`

Filter	Evaluates as
`/data IN ("zebra", "antelope", "lion")`	`true`

Working With Substrings

AMPS provides a function, `SUBSTR`, that can be used for returning a subset of a string. There are two forms of this function.

The first form takes the source string and the position at which to begin the substring. You can use a negative number to count backward from the end of the string. AMPS starts at the position specified, and returns a string that starts at the specified position and goes to the end of the string.

For example, the following expressions are all `TRUE`:

```
SUBSTR("fandango", 4) == "dango"

SUBSTR("fandango", 1) == "fandango"

SUBSTR("fandango", -2) == "go"
```

The second form of `SUBSTR` takes the source string, the position at which to begin the substring, and the length of the substring. Notice that `SUBSTR` considers the first character in the string to be position `1` (rather than position `0`), as demonstrated below. For example, the following expressions are all `TRUE`:

```
SUBSTR("fandango", 1, 3) == "fan"

SUBSTR("fandango", -4, 2) == "an"

SUBSTR("fandango", -8, 8) == "fandango"
```

Timestamp Function

AMPS includes a function that returns the current Unix timestamp.

Table 5.9. AMPS Date and Time functions

Function	Parameters	Description
UNIX_TIMESTAMP()	none	Returns the current timestamp as a double.

Geospatial Functions

AMPS includes a function for calculating the distance from a signed latitude and longitude.

Table 5.10. AMPS Geospatial Functions

Function	Parameters	Description
GEO_DISTANCE()	*first_latitude*, *first_longitude*,*second_latitude*, *second_longitude*	Returns a double that contains the distance between the point identified by *first_latitude*, *first_longitude* and

Function	Parameters	Description

		second_latitude, *second_longitude* in meters.
		
		For example, given a home point, you could use the following expression to calculate the distance from home.
		
		`GEO_DISTANCE(`
		`     /lat,`
		`     /long,`
		`     40.786995,`
		`     -119.204308)`

Chapter 6. Regular Expressions

AMPS supports regular expression matching on topics and within content filters. Regular expressions are implemented in AMPS using the Perl-Compatible Regular Expressions (PCRE) library. For a complete definition of the supported regular expression syntax, please refer to:

http://perldoc.perl.org/perlre.html

6.1. Examples

Here is an example of a content filter for messages that will match any message meeting the following criteria:

- Regular expression match of symbols of 2 or 3 characters starting with “IB”
- Regular expression match of prices starting with “90”
- Numeric comparison of prices less than 91

and, the corresponding content filter:

```
(/FIXML/Order/Instrmt/@Sym LIKE "^IB.?$") AND (/FIXML/
   Order/@Px LIKE "^90\..*" AND /FIXML/Order/@Px < 91.0)
```

Example 6.1. Filter Regular Expression Example

The tables below (Table 6.1, Table 6.2, and Table 6.3) contain a brief summary of special characters and constructs available within regular expressions.

Here are more examples of using regular expressions within AMPS.

Use `(?i)` to enable case-insensitive searching. For example, the following filter will be true regardless if `/client/country` contains “US” or “us”.

```
(/client/country LIKE "(?i)ˆus$")
```

Example 6.2. Case Insensitive Regular Expression

To match messages where tag 55 has a `TRADE` suffix, use the following filter:

```
(/55 LIKE "TRADE$")
```

Example 6.3. Suffix Matching Regular Expression

To match messages where tag 109 has a `US` prefix and a `TRADE` suffix, with case insensitive matching, use the following filter:

```
(/109 LIKE "(?i)^US.*TRADE$")
```

Example 6.4. Case Insensitive Prefix and Suffix Regular Expression

Table 6.1. Regular Expression Meta-characters

Characters	Meaning
^	Beginning of string
$	End of string
.	Any character except a newline
*	Match previous 0 or more times
+	Match previous 1 or more times
?	Match previous 0 or 1 times
|	The previous is an alternative to the following
()	Grouping of expression
[]	Set of characters
{}	Repetition modifier
\	Escape for special characters

Table 6.2. Regular Expression Repetition Constructs

Construct	Meaning
*a**	Zero or more *a*'s
a+	One or more *a*'s
a?	Zero or one *a*'s
a{*m*}	Exactly *m* *a*'s
a{*m*,}	At least *m* *a*'s
a{*m*,*n*}	At least *m*, but no more than *n* *a*'s

Table 6.3. Regular Expression Behavior Modifiers

Modifier	Meaning
i	Case insensitive search
m	Multi-line search
s	Any character (including newlines) can be matched by a . character
x	Unescaped white space is ignored in the pattern.
A	Constrain the pattern to only match the beginning of a string.
U	Make the quantifiers non-greedy by default (the quantifiers are greedy and try to match as much as possible by default.)

Raw Strings

AMPS additionally provides support for raw strings which are strings prefixed by an 'r' or 'R' character. Raw strings use different rules for how a backslash escape sequence is interpreted by the parser.

In the example below, the raw string - noted by the `r` character in the second operand of the `LIKE` predicate (Example 6.5) - will cause the results to parse the same as example which does not implement the raw string in the "LIKE" predicate (Example 6.6). In this example we are querying for string that contains the programming language named C++. In the regular string, we are required to escape the `'+'` character since it is also used by AMPS as the "match previous 1 or more times" regular expression character. In the raw string we can use `r'C++'` to search for the string and not have to escape the special `'+'` character.

```
/FIXML/Language LIKE r'C++'
```

Example 6.5. Raw String Example

```
/FIXML/Language LIKE 'C\+\+'
```

Example 6.6. Regular String Example

Topic Regular Expressions

As mentioned previously, AMPS supports regular expression filtering for topics, in addition to content filters. Regular expressions use the same grammar described in content filtering. Regular expression matching for topics is enabled in an AMPS instance by default.

Subscriptions or queries that use a regular expression for the topic name provide all matching records from AMPS topics where the name of the topic matches the regular expression used for the subscription or query. For example, if your AMPS configuration has three SOW topics, `Topic_A`, `Topic_B` and `Topic_C` and you wish to search for all messages in all of your SOW topics for records where the `Name` field is equal to "Bob", then you could use a `sow` command with a topic of `Topic_.*` and a filter of `/FIXML/@Name='Bob'` to return all matching messages that match the filter in all of the topics that match the topic regular expression.

> Results returned when performing a topic regular expression query will follow "configuration order" — meaning that the topics will be searched in the order that they appear in your AMPS configuration file. Using the above query example with `Topic_A`, `Topic_B` and `Topic_C`, if the configuration file has these topics in that exact order, the results will be returned first from `Topic_A`, then from `Topic_B` and finally the results from `Topic_C`. As with other queries, AMPS does not make any guarantees about the ordering of results within any given topic query.

Chapter 7. State of the World (SOW)

One of the core features of AMPS is the ability to persist the most recent update for each message matching a topic. The State of the World can be thought of as a database where messages published to AMPS are filtered into topics, and where the topics store the latest update to a message. Since AMPS subscriptions are based on the combination of topics and filters, the State of the World (SOW) gives subscribers the ability to quickly resolve any differences between their data and updated data in the SOW by querying the current state of a topic, or any set of messages inside a topic.

AMPS also provides the ability to keep historical snapshots of the contents of the State of the World, which allows subscribers to query the contents of the SOW at a particular point in time and replay changes from that point in time.

To provide a SOW for a topic, you must configure a SOW for the topic in the AMPS configuration file.

7.1. How Does the State of the World Work?

Much like a relational database, AMPS SOW topics contain the ability to persist the most recent update for each message. AMPS identifies a message by using a unique key for the message. The SOW key for a message is similar to the primary key in a relational database: each value of the key is a unique message. The first time a message is received with a particular SOW key, AMPS adds the message to the SOW. Subsequent messages with the same SOW key value update the message.

AMPS assigns a SOW key based on the content of the message. The fields to use for the key are specified in the SOW topic definition, and consist of one or more XPath expressions. AMPS finds the specified fields in the message and computes a SOW key based on the name of the topic and the values in these fields.

The following diagrams demonstrate how the SOW works.

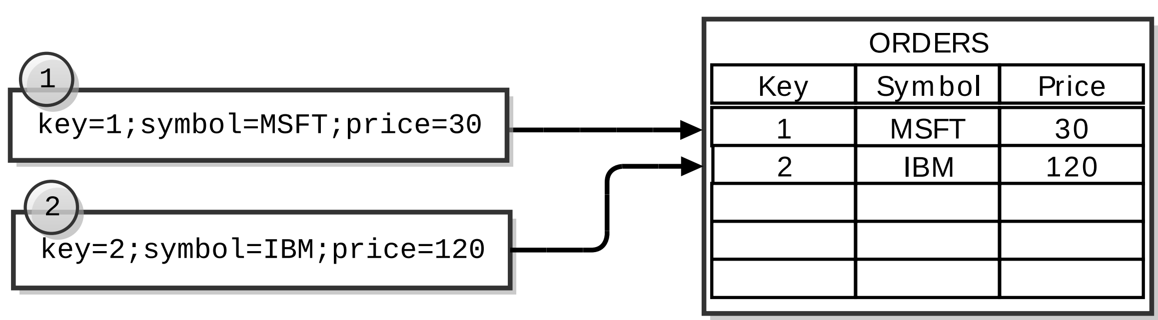


Figure 7.1. A SOW topic named `ORDERS` with a key definition of `/Key`

In Figure 7.1, two messages are published where neither of the messages have matching keys existing in the `ORDERS` topic, the messages are both inserted as new messages. Some time after these messages are processed, an update comes in for the `MSFT` order changing the price from 30 to 35. Since the `MSFT` order update has a key field of 1, this matches an existing record and overwrites the existing message containing the same key, as seen in Figure 7.2.

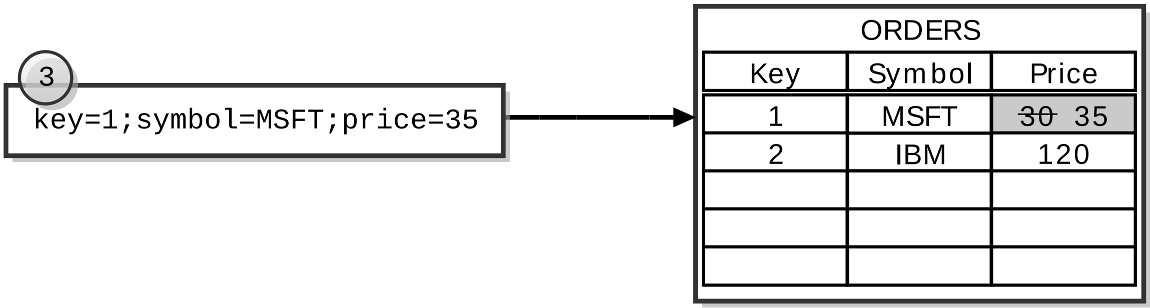


Figure 7.2. Updating the `MSFT` record by matching incoming message keys

By default, state of the world topics are *persistent*. For persistent topics, AMPS stores the contents of the state of the world in a dedicated, memory-mapped file. This means that the total state of the world does not need to fit into memory, and that the contents of the state of the world database are maintained across server restarts. You can also define a *transient* state of the world topic, which does not store the contents of the SOW to a persisted file.

The state of the world file is separate from the transaction log, and you do not need to configure a transaction log to use a SOW. When a transaction log is present that covers the SOW topic, on restart AMPS uses the transaction log to keep the SOW up to date. When the latest transaction in the SOW is more recent than the last transaction in the transaction log (for example, if the transaction log has been deleted), AMPS takes no action. If the transaction log has newer transactions than the SOW, AMPS replays those transactions into the SOW to bring the SOW file up to date. If the SOW file is missing or damaged, AMPS rebuilds the state of the world by replaying the transaction log from the beginning of the log.

When the state of the world is transient, AMPS does not store the state of the world across restarts. In this case, AMPS does not synchronize the state of the world with the transaction log when the server starts. Instead, AMPS tracks the state of the world for messages that occur while the server is running, without replaying previous messages into the SOW.

7.2. Queries

At any point in time, applications can issue SOW queries to retrieve all of the messages that match a given topic and content filter. When a query is executed, AMPS will test each message in the SOW against the content filter specified and all messages matching the filter will be returned to the client. The topic can be a literal topic name or a regular expression pattern. For more information on issuing queries, please see the *SOW Queries* chapter in the *AMPS User Guide*.

7.3. SOW Keys

This section describes AMPS SOW keys in detail, including information on how AMPS generates SOW keys and considerations for applications that generate SOW keys. An individual SOW topic may use either AMPS-generated SOW keys or user-generated SOW keys. Every message in the SOW must use the same type of key generation.

AMPS-Generated SOW Keys

AMPS-generated SOW keys are often the easiest and most reliable way to define the SOW key for a message. The advantages of this approach are that AMPS handles all of the mechanics of generating the key, the key will always match the data in the message, and there is no need for a publisher to be concerned with how AMPS assigns the key. The publisher simply publishes messages, and AMPS handles all of the details.

For example, if your SOW tracks unique orders that are identified by an `orderId` field in the message, you could provide the following Key element in your SOW configuration:

```
<Key>/orderId</Key>
```

This configuration item tells AMPS to use that field of the message to generate SOW keys.

When AMPS generates a key, it creates the key based on the *key domain* (which is the name of the topic by default) and the values of the fields specified as SOW keys. AMPS concatenates these values together with a unique separator and then calculates a checksum over the value. This ensures that different values create different keys, and ensures that records in different topics have different keys.

In some cases, you may need AMPS to calculate consistent SOW key values for identical messages even when the messages are published to different topics. The SOW topic definition allows to you to set an explicit key domain in the configuation, which AMPS will use instead of the topic name when generating SOW keys. For example, if your application uses the `orderId` field of a message as a SOW key in both a `ShippingStatus` topic and a `OpenOrders` topic, having AMPS generate a consistent key for the same `orderId` value may make it easier to correlate messages from those topics in your application. By setting the same `KeyDomain` value in the Topic configuration for those SOW topics, you can ensure that AMPS generates consistent SOW keys for the same order ID across topics.

User-Generated SOW Keys

AMPS allows applications to explicity generate and assign SOW keys. In this case, the publisher calculates the SOW key for the message and includes that key on the message when it is published. AMPS does not interpret the data in the message to decide whether the message is unique: AMPS uses only the value of the SOW key.

When using a user-generated SOW key, applications should consider the following:

- All publishers should use a consistent method for generating SOW Keys
- SOW Keys must contain only characters that are valid in Base64 encoding
- The application must ensure that messages intended to be logically different do not receive the same SOW key

User-generated SOW keys are particularly useful for the `binary` message type. For this message type, AMPS does not parse the message, so providing an explicit SOW key allows you to create a SOW that contains only `binary` messages.

7.4. SOW Indexing

AMPS maintains indexes over SOW topics to improve query efficiency. There are two types of indexes available:

- Memo indexes are created implicitly when a query uses a particular key. These indexes maintain the value of a key, and can be used for any type of query, including regular expression queries, range queries, and comparisons such as less than or greater than. You can also request that AMPS pre-create an index of this type with the `Index` directive of the SOW topic configuration.

- Hash indexes are defined by the SOW configuration. These indexes maintain a hash derived from the values provided for the fields in the key. AMPS automatically creates a hash index that contains the fields in the SOW Key. You can create any number of hash indexes for a SOW topic, with any combination of fields. Hash index queries are significantly faster than memo indexes.

  The values of hash indexes are always evaluated as strings. Hash indexes are only used for exact matches on the value of the fields and for queries that use the exact set of fields in the hash index. For example, if your configuration specifies a hash index that uses the fields `/address/postalCode` and `/customerType`, a filter such as `/address/postalCode = '99705' AND /customerType = 'retail'` will use the hash index. A filter such as `/address/postalCode = '99705' AND /customerType IN ('retail', 'remainder')` will not use the hash index, since this filter uses the `IN` operator rather than exact matching.

AMPS uses a hash index for filters wherever possible. If there is no hash index that includes exactly the keys specified in the filter, or if the filter uses operations other than equality comparison, AMPS uses a memo index if one is available. If no memo index is available, AMPS creates one during the query.

If your application frequently uses queries for an exact match on a specific set of fields (for example, retrieving a set of customers by the `/address/postalCode` field), creating a hash index can significantly improve the speed of those queries.

7.5. Configuration

Topics where SOW persistence is desired can be individually configured within the `SOW` section of the configuration file. Each topic will be defined with a `Topic` section enclosed within `SOW`. The *AMPS Configuration Reference* contains a description of the attributes that can be configured per topic. `TopicMetaData` is a synonym for `SOW` provided for compatibility with previous versions of AMPS. Likewise, `TopicDefinition` is a synonym for the `Topic` element of the SOW section, provided for compatibility with versions of AMPS prior to 5.0.

Table 7.1. SOW/Topic

Element	Description
`FileName`	The file where the State of the World (SOW) data will be stored. This element is required for SOW topics with a `Durability` of `persistent` (the default) because those topics are persisted to the filesystem. This is not required for SOW topics with a durability of `transient`.
`MessageType`	Type of messages to be stored. To use AMPS generated SOW keys, the message type specified must support content filtering so that AMPS can determine the SOW key for the message. All of the default message types, except `binary`, support content filtering. Since the `binary` message type does not support content filtering, that type can only be used for a SOW when publishers use explict keys. See the "Message Types" chapter in the *AMPS User Guide* for a discussion of the message types that AMPS loads by default. Some message types (such as Google Protocol Buffers) require additional configuration, and must be configured before using the message type in a SOW topic.
`Name`	The name of the SOW topic - all unique messages (see `Key`) on this topic will be stored in a topic-specific SOW database.

Element	Description
	If no Name is provided, AMPS accepts Topic as a synonym for Name to provide compatibility with versions of AMPS previous to 5.0.
Key	Specifies an XPath within each message that AMPS will use to generate a SOW key, which determines whether a message is unique. This element can be specified multiple times to create a composite key from the combined value of the specified Key elements. When one or more Key elements is specified for the SOW, AMPS generates the SOW key for each message. When no Key field is specified, publishers must explicitly provide the SOW key for each message published to this topic. 60East recommends configuring a Key and having AMPS generate the SOW key for a message unless your application has specific needs that make this impractical. AMPS automatically creates a hash index for the set of fields specified in the Key elements.
HashIndex	AMPS provides the ability to do fast lookup for SOW records based on specific fields. When one or more HashIndex elements are provided, AMPS creates a hash index for the fields specified in the element. These indexes are created on startup, and are kept up to date as records are added, removed, and updated. The HashIndex element contains a Key element for each field in the hash index. AMPS uses a hash index when a query uses a exact matching for all of the fields in the index. AMPS does not use hash indexes for range queries or regular expressions. AMPS automatically creates a hash index for the set of fields specified in the set of Key fields for the SOW, if those fields are specified.
RecoveryPoint	For SOW topics that are covered by the transaction log, the point from which to recover the SOW if the SOW file is removed, or if the SOW topic has transient duration. This configuration item allows two values: • epoch recovers the SOW from the beginning of the transaction log • now recovers the SOW from the current point in the transaction log Defaults to epoch.
Index	AMPS supports the ability to precreate memo indexes for specific fields using the Index configuration option. When one or more Index elements are provided, AMPS creates memo indexes for any field specified in an Index element on startup, before a query that uses that field runs. Otherwise, AMPS indexes each field the first time a query uses the field. Adding one or more Index configurations to a SOW/Topic can improve retrieval performance the first time a query that contains the indexed fields runs for large SOW topics.
SlabSize	The size of each allocation for the SOW file, as a number of bytes. When AMPS needs more space for the SOW, it requests this amount of space from the operating system. This effectively sets the maximum message size that AMPS guarantees can be stored in the SOW.

Element	Description
	60East recommends setting this value only if you will be storing messages larger than the default `SlabSize` or if performance or capacity testing indicates a need to tune SOW performance. If you plan to store messages larger than the default setting, 60East recommends a starting value of several times the maximum message size. For example, if your maximum message size is 2MB, a good starting point for `SlabSize` would be 8MB. If it becomes necessary to tune the `SlabSize`, see the *Best Practices* and *Capacity Planning* sections of the AMPS User Guide for a full discussion tuning the `SlabSize`. Default: `1MB`
`InitialSlabCount`	The number of SOW slabs that AMPS will allocate on startup. Default: `1` Maximum: `1024`
`Expiration`	Time for how long a record should live in the SOW database for this topic. The expiration time is stored on each message, so changing the expiration time in the configuration file will not affect the expiration of messages currently in the SOW. AMPS accepts interval values for the Expiration, using the interval format described in the AMPS Configuration Guide section on units, or one of the following special values: • A value of `disabled` specifies that AMPS will not process SOW expiration for this topic, regardless of any expiration value set on the message. In this case, AMPS saves the expiration for the message, but does not process it. The value must be set to `disabled` (the default) if `History` is enabled for this topic. • A value of `enabled` specifies that AMPS will process SOW expiration for this topic, with no expiration set by default. Instead, AMPS uses the value set on the individual messages (with no expiration set for messages that do not contain an expiration value). Default: `disabled` (never expire)
`KeyDomain`	The seed value for `SowKeys` used within the topic. The default is the topic name, but it can be changed to a string value to unify `SowKey` values between different topics. For example, if your application has a `ShippingAddress` SOW and a `CreditRating` SOW that both use `/customerID` as the SOW key, you can use a `KeyDomain` to ensure that the generated `SowKey` for a given `/customerId` is identical for both SOW topics. This does not affect how AMPS processes the SOW topics, but can make correlating information from different SOW topics easier in your application. Default: the name of the SOW topic
`Durability`	Defines the data durability of a SOW topic. SOW databases listed as `persistent` are stored to the file system, and retain their data across instance restarts. Those listed as `transient` are not persisted to the file system, and are reset each time the AMPS instance restarts. Default: `persistent` Valid values: `persistent` or `transient`

Element	Description
	Synonyms: `Duration` is also accepted for this parameter for backward compatibility with configuration prior to 4.0.0.1
`History`	Enable historical query for this SOW. This element contains a `Window` and `Granularity` element. When the `History` element is present, historical query is enabled for this sow. Otherwise, AMPS does not enable historical query and does not store the historical state of the SOW. `Expiration` must be `disabled` when `History` is enabled.
`Window`	For a historical SOW, the length of time to store history. For example, when the value is `1w`, AMPS will store one week of history for this SOW. Used within the `History` element. Default: By default, AMPS does not expire historical SOW data.
`Granularity`	For a historical SOW, the granularity of the history to store. For many applications, it is not necessary for AMPS to store all of the updates to the SOW. This parameter sets the resolution at which AMPS will save the state of a message. For example, when you set a granularity of `1m`, AMPS will save the state of the message no more frequently than once per minute, even when the state of the message is updated several times a minute. Used within the `History` element.
DEPRECATED: `RecordSize`	*This parameter is deprecated beginning in AMPS 5.0. Use the* `SlabSize` *parameter instead.* Size (in bytes) of a SOW record for this topic. Default: `512`
DEPRECATED: `InitialSize`	*This parameter is deprecated beginning in AMPS 5.0. Use the* `InitialSlabCount` *parameter instead.*Initial size (in records) of the SOW database file for this topic. Default: `2048`
DEPRECATED: `IncrementSize`	*This parameter is deprecated beginning in AMPS 5.0. Use the* `SlabSize` *parameter instead.* Number of records to expand the SOW database (for this topic) by when more space is required. Default: `1000`

The listing in Example 7.1 is an example of using `Topic` to add a SOW topic to the AMPS configuration. One topic named `ORDERS` is defined as having key `/invoice`, `/customerId` and `MessageType` of `json`. The persistence file for this topic be saved in the `sow/ORDERS.json.sow` file. For every message published to the `ORDERS` topic, a unique key will be assigned to each record with a unique combination of the fields `/invoice` and `/customerId`. A second topic named `ALERTS` is also defined with a `MessageType` of `xml` keyed off of `/client/id`. The SOW persistence file for `ALERTS` is saved in the `sow/ALERTS.xml.sow` file.

```
<SOW>
    <Topic>
        <Name>ORDERS</Name>
        <FileName>sow/%n.sow</FileName>
        <Key>/invoice</Key>
```

```
        <Key>/customerId</Key>
        <MessageType>json</MessageType>
        <SlabSize>1MB</SlabSize>
        <HashIndex>
           <Key>/region</Key>
        </HashIndex>
    </Topic>

    <Topic>
        <Name>ALERTS</Name>
        <FileName>sow/%n.sow</FileName>
        <Key>/alert/id</Key>
        <MessageType>xml</MessageType>
    </Topic>
</SOW>
```

Example 7.1. Sample SOW configuration

Topics are scoped by their message type.

For example, two topics named Orders can be created one which supports `MessageType` of `json` and another which supports `MessageType` of `xml`.

Each of the `MessageType` entries that are defined for the `Orders` topic will require that `Transport` in the configuration file can accept messages of that type. Otherwise, there is no way for a publisher to publish messages of that type to this instance or for a subscriber to receive messages of that type from this instance.

This means that messages published to the `Orders` topic must know the type of message they are sending (`json` or `xml`) and the port defined by the transport.

Chapter 8. SOW Queries

When SOW topics are configured inside an AMPS instance, clients can issue SOW queries to AMPS to retrieve all of the messages matching a given topic and content filter. When a query is executed, AMPS will test each message in the SOW against the content filter specified and all messages matching the filter will be returned to the client. The topic can be a straight topic or a regular expression pattern.

8.1. SOW Queries

A client can issue a query by sending AMPS a `sow` command and specifying an AMPS topic. Optionally a filter can be used to further refine the query results. AMPS also allows you to restrict the query to a specific set of messages identified by a set of SowKeys. When AMPS receives the `sow` command request, it will validate the filter and start executing the query. When returning a query result back to the client, AMPS will package the `sow` results into a `sow` record group by first sending a `group_begin` message followed by the matching SOW records, if any, and finally indicating that all records have been sent by terminating with a `group_end` message. The message flow is provided as a sequence diagram in Figure 8.1.

For purposes of correlating a query request to its result, each query command can specify a `QueryId`. The `QueryId` specified will be returned as part of the response that is delivered back to the client. The `group_begin` and `group_end` messages will have the `QueryId` attribute set to the value provided by the client. The client specified `QueryId` is what the client can use to correlate query commands and responses coming from the AMPS engine.

AMPS does not allow a `sow` command on topics that do not have a SOW enabled. If a client queries a topic that does not have a SOW enabled, AMPS returns an error.

> ⚠ The ordering of records returned by a SOW query is undefined by default. You can also include an OrderBy parameter on the query to specify a particular ordering based on the contents of the messages.

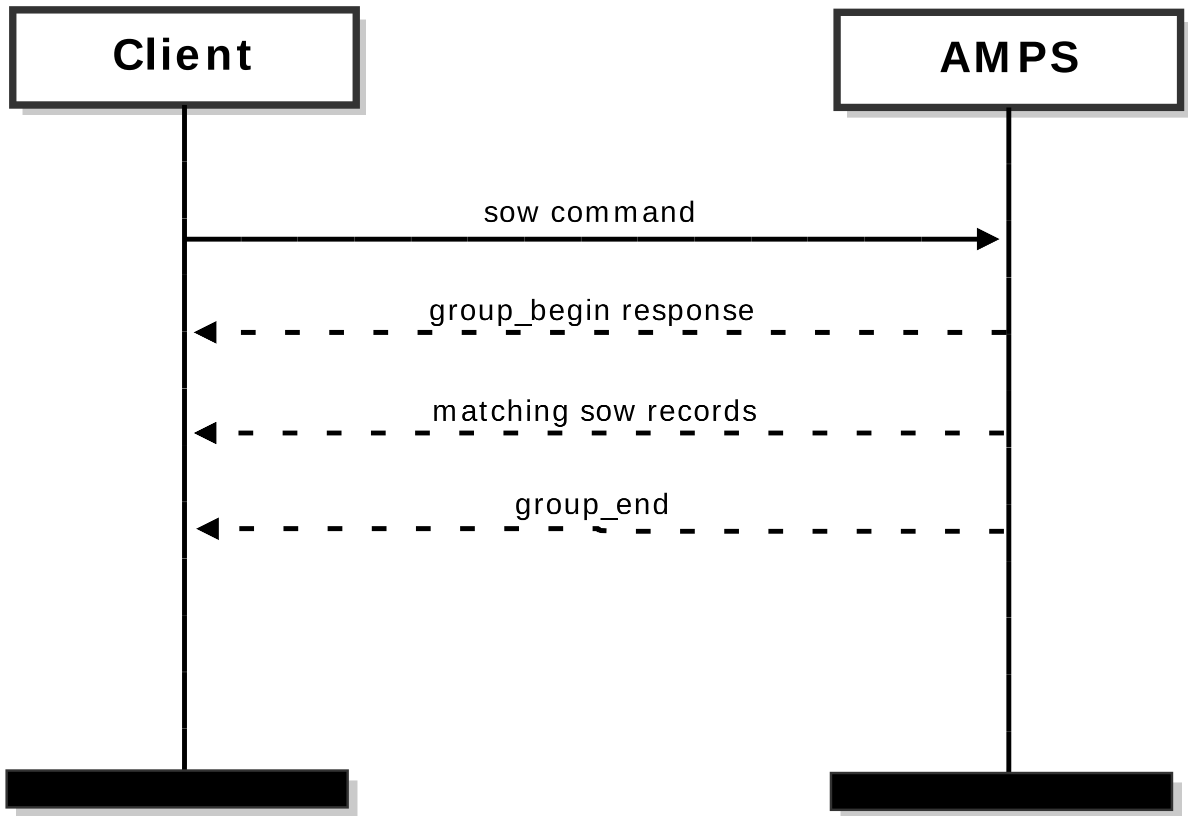


Figure 8.1. SOW Query Sequence Diagram

8.2. Historical SOW Queries

SOW topics can also be configured to include historical snapshots of messages, which allows subscribers to retrieve the contents of the SOW that reflect a particular point in time.

As with simple queries, a client can issues a query by sending AMPS a `sow` command and specifying an AMPS topic. For a historical query, the client also adds a timestamp that includes the point in time for the query. A filter can be used to further refine the query results based on the message content.

Window and Granularity

AMPS allows you to control the amount of storage to devote to historical SOW queries through the `Window` and `Granularity` configuration options.

The `Window` option sets the amount of time that AMPS will retain historical copies of messages. After the amount of time set by the `Window`, AMPS may discard copies of the messages.

The Granularity option sets the interval at which AMPS retains a historical copy of a message in the SOW. For example, if the Granularity is set to 10m, AMPS stores a historical copy of the message no more frequently than every 10 minutes, regardless of how many times the message is updated in that 10 minute interval. AMPS stores the copies when a new message arrives to update the SOW. This means that AMPS always returns a valid SOW state that reflects a published message, but -- as with a conflated topic -- the SOW may not reflect all of the states that a message passes through. This also means that AMPS uses SOW space efficiently. If no updates have arrived for a message, since the last time a historical message was saved, AMPS has no need to save another copy of the message.

When a historical SOW and Subscribe query is entered, and the topic is covered by a transaction log, AMPS returns the state of the SOW adjusted to the next oldest granularity, then replays messages from that point. In other words, AMPS returns the same results as a historical SOW query, then replays the full sequence of messages from that point forward.

Message Sequence Flow

The message sequence flow is the same as for a simple SOW query. Once AMPS has transmitted the messages that were in the SOW as of the timestamp of the query, the query ends. Notice that this replay includes messages that have been subsequently deleted from the SOW.

8.3. SOW Query-and-Subscribe

AMPS has a special command that will execute a query and place a subscription at the same time to prevent a gap between the query and subscription where messages can be lost. Without a command like this, it is difficult to reproduce the SOW state locally on a client without creating complex code to reconcile incoming messages and state.

For an example, this command is useful for recreating part of the SOW in a local cache and keeping it up to date. Without a special command to place the query and subscription at the same moment, a client is left with two options:

1. Issue the query request, process the query results, and then place the subscription, which misses any records published between the time when the query and subscription were placed; or

2. Place the subscription and then issue the query request, which could send messages placed between the subscription and query twice.

Instead of requiring every program to work around these options, the AMPS sow_and_subscribe command allows clients to place a query and get the streaming updates to matching messages in a single command.

In a sow_and_subscribe command, AMPS behaves as if the SOW command and subscription are placed at the exact same moment.The SOW query will be sent before any messages from the subscription are sent to the client. Additionally, any new publishes that come into AMPS that match the sow_and_subscribe filtering criteria and come in after the query started will be sent after the query finishes (and the query will not include those messages.)

AMPS allows a sow_and_subscribe command on topics that do not have a SOW enabled. In this case, AMPS simply returns no messages between group_begin and group_end.

The message flow as a sequence diagram for sow_and_subscribe commands is contained in Figure 8.2.

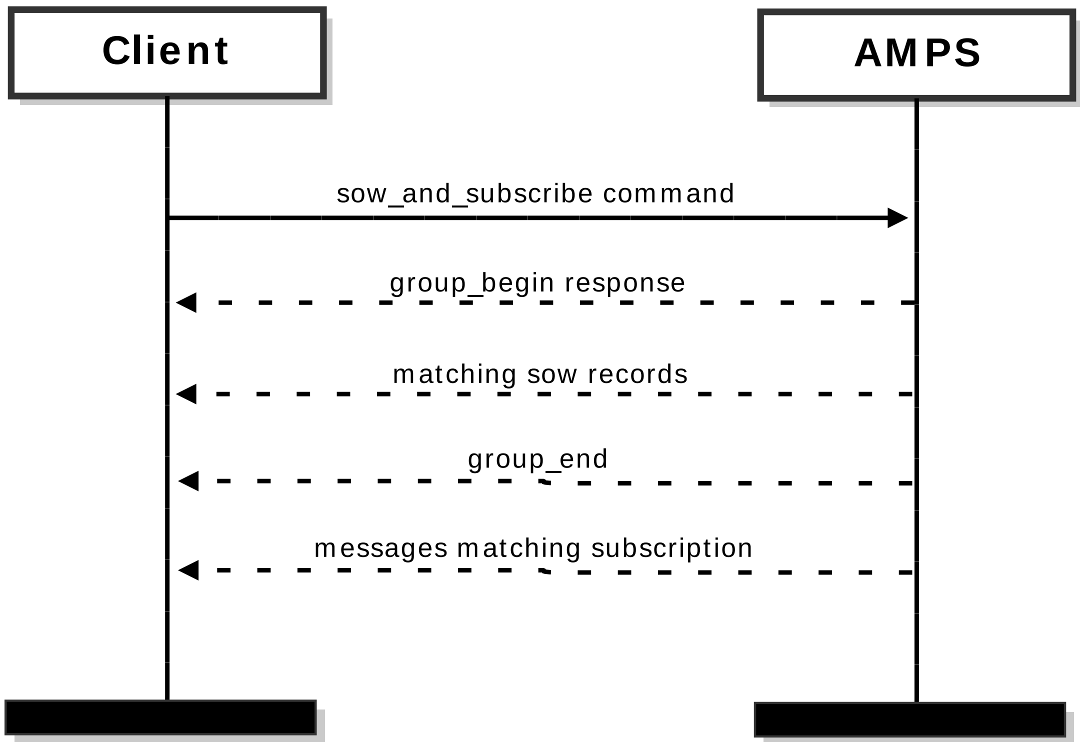


Figure 8.2. SOW-And-Subscribe Query Sequence Diagram

Historical SOW Query and Subscribe

AMPS SOW Query and Subscribe also allows you to begin the subscription with a historical SOW query. For historical SOW queries, the subscription begins at the point of the query with the results of the SOW query. The subscription then replays messages from the transaction log. Once messages from the transaction log have been replayed, the subcription then provides messages as AMPS publishes them.

In effect, a SOW Query and Subscribe with a historical query allows you to recreate the client state and processing as though the client had issued a SOW Query and Subscribe at the point in time of the historical query.

A historical SOW and subscribe requires that the SOW topic is recorded in the transaction log and that history is enabled on the SOW. If history is not enabled for the topic, a SOW and subscribe command returns the current state of the SOW and the subscription begins atomically at the point in time when AMPS processes the command.

Replacing Subscriptions with SOW and Subscribe

As described in Section 4.3, AMPS allows you to replace an existing subscription. When the subscription is a SOW and Subscribe, AMPS will re-run the SOW query delivering the messages that are in scope with the new filter but which were not previously delivered. If the subscription requests out-of-focus (OOF) messages, AMPS will deliver out of focus messages for messages that matched the previous filter but do not match the new filter. As with the initial query and subscribe, AMPS guarantees to deliver any changes to the SOW that match the filter and occur after the point of the query.

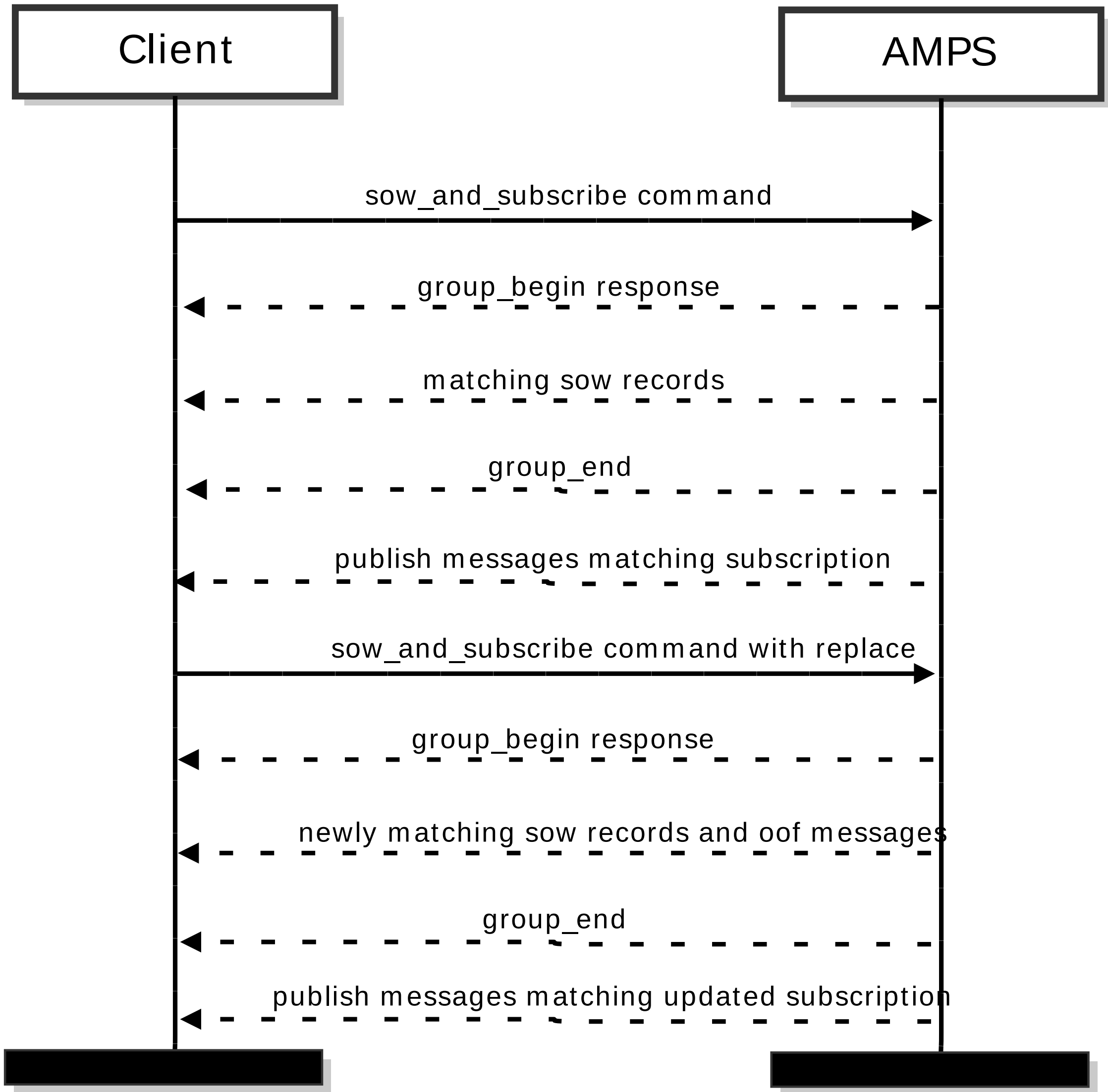


Figure 8.3. SOW And Subscribe Replace Sequence Diagram

8.4. SOW Query Response Batching

When processing a SOW query, AMPS has the ability to combine messages into batches for more efficient network usage. The maximum number of messages in a batch is determined by the `BatchSize` parameter on the SOW query command. AMPS defaults to a `BatchSize` value of 1, meaning AMPS sends one message per batch in the response. The `BatchSize` is the maximum number of records that will be returned within a single response payload. Each AMPS response for the query contains a `BatchSize` value in its header to indicate the number of messages in the batch. This number will be anywhere from 1 to `BatchSize`.

Current versions of the AMPS client libraries set a batch size of 10 when using the named convenience methods (for example, `sowAndSubscribe`) if no other batch size is specified.

Notice that the format of messages returned from AMPS may be different depending on the message type requested. However, the information contained in the messages is the same for all message types.

> When issuing a `sow_and_subscribe` command AMPS will return a `group_begin` and `group_end` segment of messages before beginning the live subscription sequence of the query. This is also true when a `sow_and_subscribe` command is issued against a non-SOW topic. In this later case, the `group_begin` and `group_end` will contain no messages.

Using a `BatchSize` greater than 1 can yield greater performance, particularly when querying a large number of small records. In general, 60East recommends using a `BatchSize` that provides good network utilization without consuming excessive server memory. Most applications use a batch size designed to create batches that fit well into the maximum transmission unit (MTU) for the network. AMPS reports an error if an application requests a batch size larger than 10,000 records (this value is orders of magnitude larger than the typical `BatchSize` used by applications).

For applications where the average message size is close to, or larger than, the MTU for the network, 60East recommends using a smaller `BatchSize`. For messages that are many times the MTU, 60East recommends a `BatchSize` of 1.

> Using an appropriate `BatchSize` parameter is critical to achieve the maximum query performance with a large number of small messages.

Each message within the batch will contain id and key values to help identify each message that is returned as part of the overall response.

For XML, the format of the response is:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header>
    <Cmd>sow</Cmd>
    <TxmTm>20080210-17:16:46.066-0500</TxmTm>
    <QId>100</QId>
    <GrpSqNum>1</GrpSqNum>
    <BtchSz>5</BtchSz>
    <Tpc>order</Tpc>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <Msg key="143101935596417" len="120"> ... </Msg>
    <Msg key="86456484160208" len="125"> ... </Msg>
    <Msg key="18307726844082" len="128"> ... </Msg>
    <Msg key="15874572074104" len="118"> ... </Msg>
    <Msg key="61711462299630" len="166"> ... </Msg>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For FIX, the format has the following form:

```
{sow header}❶{message header}❶{message data}❷{message header}❶{message
 data}❷...
```

❶ header separator
❷ message separator

Each message header will contain the `SowKey` and the `MessageLength` attributes. The `MessageLength` is intended to help clients parse the response with minimal effort. It indicates the length of the message data contained in the message.

The following is an example FIX message SOW query response message:

```
20000=sow❶20004=20080210-17:16:46.066-0500❶20007=fix
❶20019=100❶20060=1❶20023=5❶20005=order
❶❷20059=1❶20058=128❶❷fix message data❸
```

❶ header separator
❷ header separator
❸ message separator

> ⚠ Care should be taken when issuing queries that return large results. When contemplating the usage of large queries and how that impacts system reliability and performance, please see the section called “Slow Clients ” for more information.

For more information on executing queries, please see the Developer Guide for the AMPS client of your choice.

8.5. Configuring SOW Query Result Sets

AMPS allows you to control the results returned by a SOW query by including the following optional headers on the query:

Table 8.1. SOW Query Options

Option	Result
TopN	Limits the results returned to the number of messages specified.

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.

For details on how to submit these options with a SOW query, see the documentation for the AMPS client library your application uses.

Chapter 9. SOW Message Expiration

By default, SOW topics stores all distinct records until the record is explicitly deleted. For scenarios where message persistence needs to be limited in duration, AMPS provides the ability to set a time limit on the lifespan of SOW topic messages. This limit on duration is known as message expiration and can be thought of as a “Time to Live” feature for messages stored in a SOW topic.

9.1. Usage

Expiration on SOW topics is disabled by default. For AMPS to expire messages in a SOW topic, you must explicitly enable expiration on the SOW topic.

There are two ways message expiration time can be set. First, a SOW topic can specify a default lifespan for all messages stored for that SOW topic. Second, each message can provide an expiration as part of the message header.

The expiration for a given SOW topic message is first determined based on the message expiration specified in the message header. If a message has no expiration specified in the header, then the message will inherit the expiration setting for the topic expiration. If there is no message expiration and no topic expiration, then it is implicit that a SOW topic message will not expire.

Topic Expiration

AMPS configuration supports the ability to specify a default message expiration for all messages in a single SOW topic. Below is an example of a configuration section for a SOW topic definition with an expiration. Chapter 7 has more detail on how to configure the SOW topic.

```
<SOW>
  <Topic>
    <Name>ORDERS</Name>
    <FileName>sow/%n.sow</FileName>
    <Expiration>30s</Expiration>
    <Key>/55</Key>
    <Key>/109</Key>
    <MessageType>fix</MessageType>
  </Topic>
</SOW>
```

Example 9.1. Topic Expiration

In this case, messages with no lifetime specified on the message have a 30 second lifetime in the SOW. When a message arrives and that message has an expiration set, the message expiration overrides the default expiration for the topic.

AMPS also allows you to enable expiration on a SOW topic, but to only expire messages that have message-level expiration set:

```
<SOW>
  <Topic>
    <Name>ORDERS</Name>
```

```
    <FileName>sow/%n.sow</FileName>
    <Expiration>enabled</Expiration>
    <Key>/55</Key>
    <Key>/109</Key>
    <MessageType>fix</MessageType>
  </Topic>
</SOW>
```

Example 9.2. Topic Expiration

With this configuration file, expiration is enabled for the topic. The message lifetime is specified on each individual message. When expiration is disabled for a SOW topic, AMPS preserves any message expiration set on an individual message but does not expire messages.

AMPS processes expirations during startup when SOW expiration is enabled. This means that any record in the SOW which needs to be expired will be expired as AMPS starts. Notice that if the expiration period has changed in the configuration file (or expiration has been enabled or disabled), AMPS processes the SOW using the current expiration configuration. For messages that were not published with an explicit expiration, the lifetime defaults to the current expiration period for the topic.

Message Expiration

Individual messages have the ability to specify an expiration for a a published message. Below is an example of an XML message that is publishing an `Order`, and has an expiration set for 20 seconds.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header>
    <Cmd>publish</Cmd>
    <TxmTm>20061201-17:29:12.000-0500</TxmTm>
    <Expn>20</Expn>
    <Tpc>order</Tpc>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <FIXML>
      <Order Side="2" Px="32.00">
        <Instrmt Sym="MSFT"/>
        <OrdQty Qty="100"/>
      </Order>
    </FIXML>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 9.3. Message Expiration

9.2. Example Message Lifecycle

When a message arrives, AMPS calculates the expiration time for the message and stores a timestamp at which the message expires in the SOW with the message. When the message contains an expiration time, AMPS uses that time to create the timestamp. When the message does not include an expiration time, if the topic contains an expiration time, AMPS uses the topic expiration for the message. Otherwise, there is no expiration set on the message, and AMPS records a timestamp value that indicates no expiration.

Messages in the SOW topic can receive updates before expiration. When a message is updated, the message's expiration lifespan is reset. For example, a message is first published to a SOW topic with an expiration of 45 seconds. The message is updated 15 seconds after publication of the initial message, and the update resets the expiration to a new 45 second lifespan. This process can continue for the entire lifespan of the message, causing a new 45 second lifespan renewal for the message with every update.

If a message expires, then the message is deleted from the SOW topic. This event will trigger delete processing to be executed for the message, similar to the process of executing a **sow_delete** command on a message stored in a SOW topic.

Recovery and Expiration

When using message expiration, one common scenario is that the message has an expiration set, but the AMPS instance is shut down during the lifetime of the message.

To handle such a scenario, AMPS calculates and stores a timestamp for the expiration, as described above. Therefore, if the AMPS instance is shutdown, then upon recovery the engine will check to see which messages have expired since the occurrence of the shutdown. Any expired messages will be deleted as soon as possible.

Notice that, because the timestamp is stored with each message, changing the default expiration of a SOW topic does not affect the lifetime of messages already in the SOW. Those timestamps have already been calculated, and AMPS does not recalculate them when the instance is restarted or when the defaults on the SOW topic change. If expiration is not enabled for the topic after the configuration change, AMPS does not process expirations for that topic and messages will not expire.

Chapter 10. Out-of-Focus Messages (OOF)

One of the more difficult problems in messaging is knowing when a record that previously matched a subscription has been updated so that the record no longer matches the subscription. AMPS solves this problem by providing an out-of-focus, or *OOF*, message to let subscribers know that a record they have previously received no longer matches the subscription. The OOF messages help subscribers easily maintain state and remove records that are no longer relevant.

OOF notification is optional. A subscriber must explicitly request that AMPS provide out-of-focus messages for a subscription.

When OOF notification has been requested, AMPS produces an `oof` message for any record that has previously been received by the subscription at the point at which:

- The record is deleted,
- The record expires,
- The record no longer matches the filter criteria, *or*
- The subscriber is no longer entitled to view the new state of the record

AMPS produces an `oof` message for each record that no longer matches the subscription. The `oof` message is sent as part of processing the update that caused the record to no longer match. Each `oof` message contains information the subscriber can use to identify the record that has gone out of focus and the reason that the record is now out of focus.

Because AMPS must maintain the current state of a record to know when to produce an `oof` message, these messages are only supported for SOW topics.

10.1. Usage

Consider the following scenario where AMPS is configured with the following SOW key for the buyer topic:

```
<SOW>
  <Topic>
    <Name>buyer</Name>
    <MessageType>xml</MessageType>
    <Key>/buyer/id</Key>
  </Topic>
</SOW>
```

Example 10.1. Topic Configuration

When the following message is published, it is persisted in the SOW topic:

```
<buyer><id>100</id><loc>NY</loc></buyer>
```

Example 10.2. First Publish Message

A client issues a `sow_and_subscribe` request for the topic `buyer` with the filter `/buyer/loc="NY"` and the `oof` option set on the request. The client will be sent the messages as part of the SOW query result.

Subsequently, the following message is published to update the `loc` tag to LN:

```
<buyer><id>100</id><loc>LN</loc></buyer>
```

Example 10.3. Second Publish Message

The original message in the SOW cache is updated. The client does not receive the second publish message, because that message does not match the filter (`/buyer/loc="NY"`). This is problematic. The client has a message that is no longer in the SOW cache and that no longer matches the current state of the record. Because the `oof` option was set on the subscription, however, the AMPS engine sends an `oof` message to let these clients know that the message that they hold is no longer in the SOW cache. The following is an example of what's returned:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header>
    <Reason>match</Reason>
    <Tpc>buyer</Tpc>
    <Cmd>oof</Cmd>
    <MsgTyp>xml</MsgTyp>
    <SowKey>6387219447538349146</SowKey>
    <SubIds>SAMPS-1214725701_1</SubIds>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <client>
      <id>100</id>
      <loc>LN</loc>
    </client>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 10.4. oof xml example message

An easy way to think about the situations where AMPS sends an OOF notification is to consider what would happen if the client re-issued the original `sow` request after the above message was published. The `/client/loc="NY"` expression no longer matches the message in the SOW cache and as a result, this message would not be returned.

When AMPS returns an OOF message, the data contained in the body of the message represents the updated state of the OOF message (except as described below). This will allow the client to make a determination as to how to handle the data, be it to remove the data from the client view or to change their query to broaden the filter thresholds. This enables a client to take a different action depending on why the message no longer matches. For example, an application may present a different icon for an order that moves to a status of `completed` than it would present for an order that moves to a status of `cancelled`.

When a `delta_publish` message causes the SOW record to go out of focus, AMPS returns the merged record.

When there is no updated message to send, AMPS sends the state of the record before the change that produced the OOF. This can occur when the message had been deleted, when the message has expired, or when an update causes the client to no longer have permission to receive the record.

10.2. Example

To help reinforce the concept of OOF messages, and how OOF messaging can be used in AMPS, consider a scenario where there is a GUI application whose requirement is to display all open orders of a client. There are several possible solutions to ensure that the GUI client data is constantly updated as information changes, some of which are examined below; however, the goal of this section is to build up a `sow_and_subscribe` message to demonstrate the power that OOF notifications add to AMPS.

Client-Side Filtering in a sow_and_subscribe Command

First, consider an approach that sends a `sow_and_subscribe` message on the topic `orders` using the filter `/Client="Adam"`:

AMPS completes the `sow` portion of this call by sending all matching messages from the `orders` SOW topic. AMPS then places a subscription whereby all future messages that match the filter get sent to the subscribing GUI client.

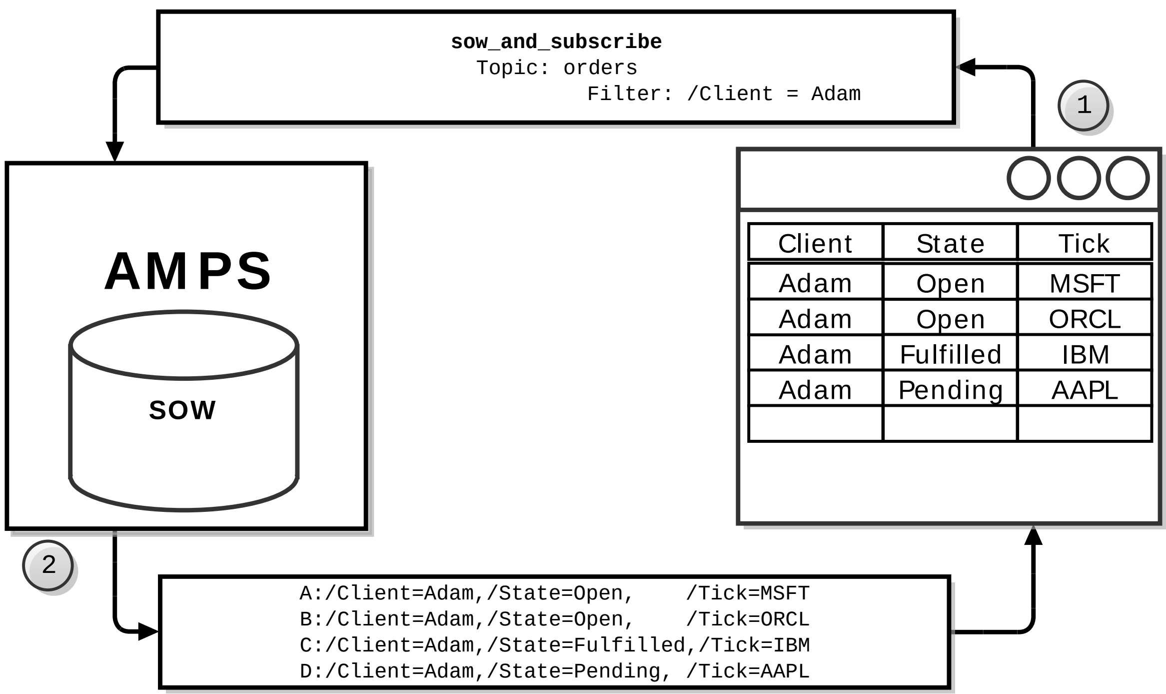


Figure 10.1. `sow_and_subscribe` example

As the messages come in, the GUI client will be responsible for determining the state of the order. It does this by examining the `State` field and determining if the state is equal to “Open” or not, and then updating the GUI based on the information returned.

This approach puts the burden of work on the GUI and, in a high volume environment, has the potential to make the client GUI unresponsive due to the potential load that this filtering can place on a CPU. If a client GUI becomes unresponsive, AMPS will queue the messages to ensure that the client is given the opportunity to catch up. The specifics of how AMPS handles slow clients is covered in the section called “Slow Clients ”.

AMPS Filtering in a sow_and_subscribe command

The next step is to add an additional 'AND' clause to the filter. In this scenario we can let AMPS do the filtering work that was previously handled on the client. This is accomplished by modifying our original `sow_and_subscribe` to use the following filter:

```
/Client = "Adam" AND /State = "Open"
```

Similar to the above case, this `sow_and_subscribe` will first send all messages from the `orders` SOW topic that have a `Client` field matching "Adam" and a `State` field matching "Open." Once all of the SOW topic messages have been sent to the client, the subscription will ensure that all future messages matching the filter will be sent to the client.

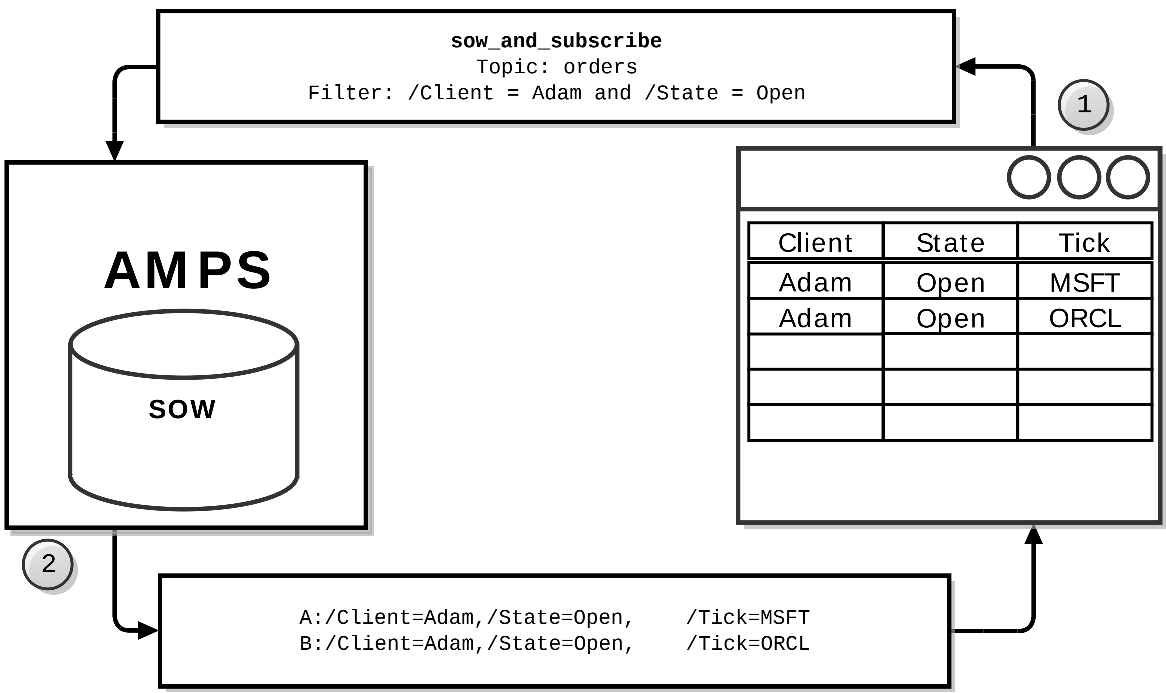


Figure 10.2. State Filter in a sow_and_subscribe

There is a less obvious issue with this approach to maintaining the client state. The problem with this solution is that, while it initially will yield all open orders for client "Adam", this scenario is unable to stay in sync with the server. For example, when the order for Adam is filled, the `State` changes to `State=Filled`. This means that, inside AMPS, the order on the client will no longer match the initial filter criteria. The client will continue to display and maintain these out-of-sync records. Since the client is not subscribed to messages with a `State` of "Filled," the GUI client would never be updated to reflect this change.

OOF Processing in a sow_and_subscribe command

The final solution is to implement the same `sow_and_subscribe` query which was used in the first scenario. This time, we use the filter requests only the State that we're interested in, but we add the `oof` option to the command so the subscriber receives OOF messages.

```
/Client = "Adam" AND /State = "Open"
```

AMPS will respond immediately with the query results, in a similar manner to a `sow_and_subscribe` (Figure 10.3) command.

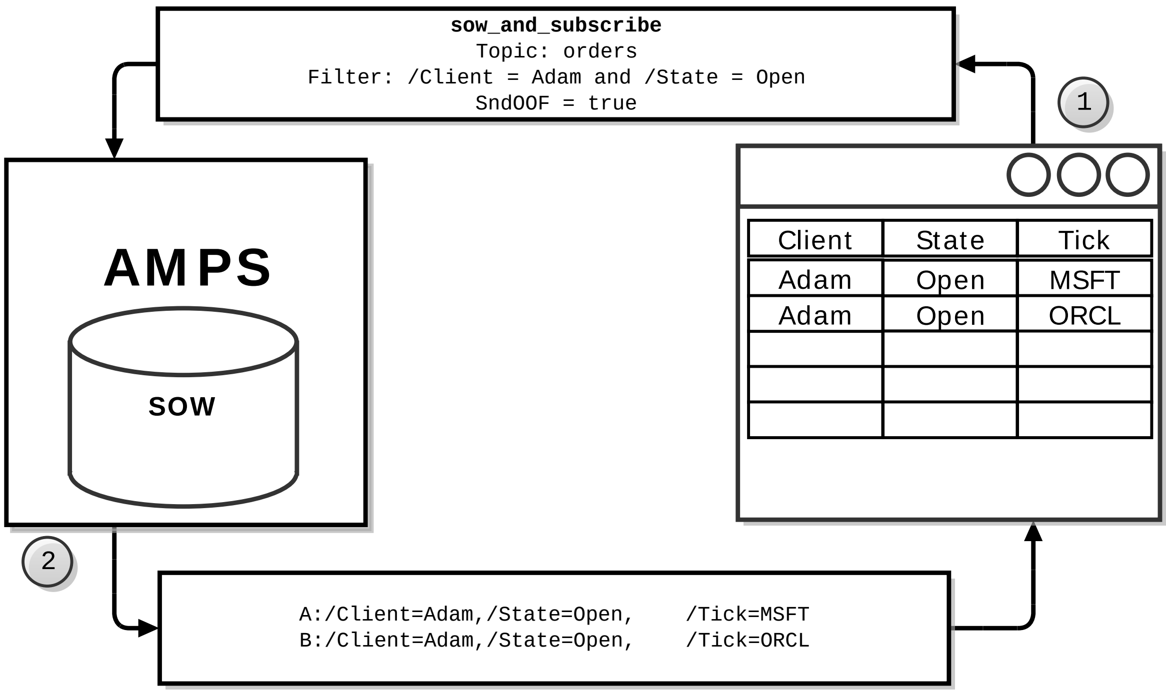


Figure 10.3. `sow_and_subscribe` with `oof` enabled

This approach provides the following advantage. For all future messages in which the same `Open` order is updated, such that its status is no longer `Open`, AMPS will send the client an `OOF` message specifying that the record which previously matched the filter criteria has fallen out of focus. AMPS will not send any further information about the message unless another incoming AMPS message causes that message to come back into focus.

In Figure 10.4 the Publisher publishes a message stating that Adam's order for MSFT has been fulfilled. When AMPS processes this message, it will notify the GUI client with an `OOF` message that the original record no longer matches the filter criteria. The `OOF` message will include a `Reason` field with it in the message header, defining the reason for the message to lose focus. In this case the `Reason` field will state `match` since the record no longer matches the filter

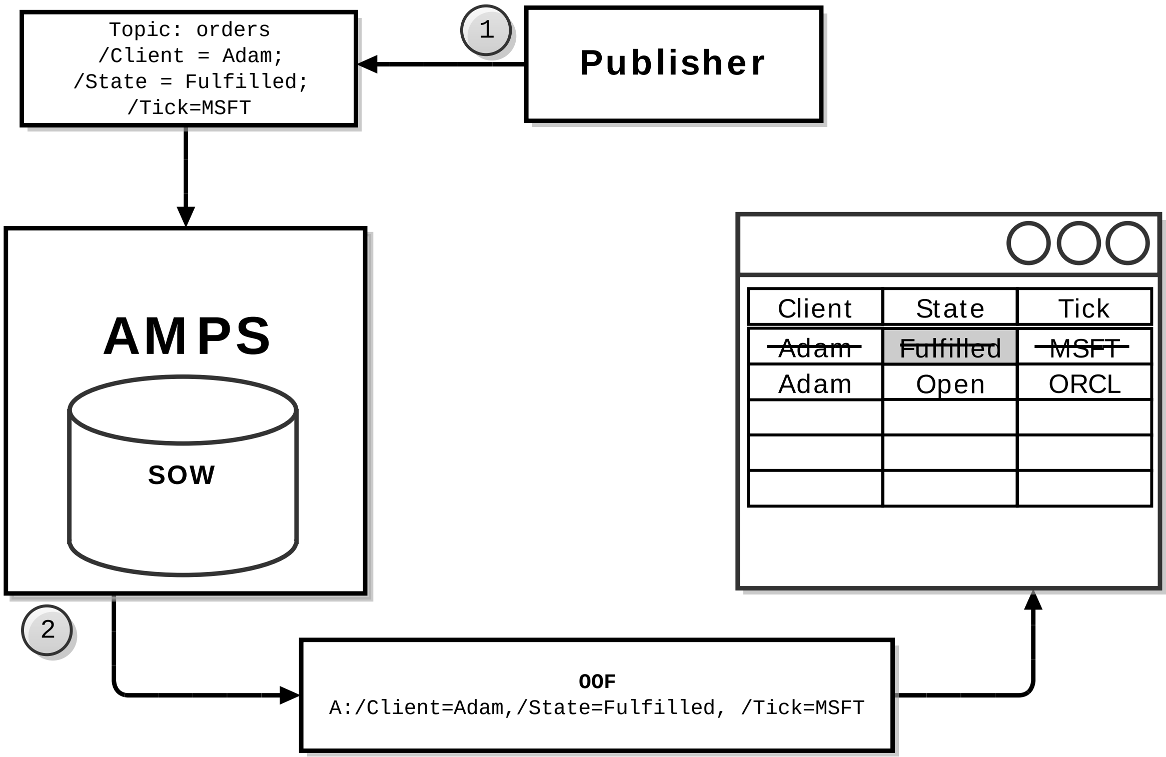


Figure 10.4. OOF message

AMPS will also send `OOF` messages when a message is deleted or has expired from the SOW topic.

We see the power of the `OOF` message when a client application wants to have a local cache that is a subset of the SOW. This is best managed by first issuing a query filter `sow_and_subscribe` which populates the GUI, and enabling the `oof` option. AMPS informs our application when those records which originally matched no longer do, at which time the program can remove them.

Chapter 11. Delta Messaging

AMPS *delta messaging* allows applications to work with only the changed parts of a message in the SOW. In high-performance messaging, it's important that applications not waste time or bandwidth for messages that they aren't going to use.

Delta messaging has two distinct aspects:

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

While these features are often used together, the features are independent. For example, a subscriber can request a regular subscription even if a publisher is publishing deltas. Likewise, a subscriber can request a delta subscription even if a publisher is publishing full messages.

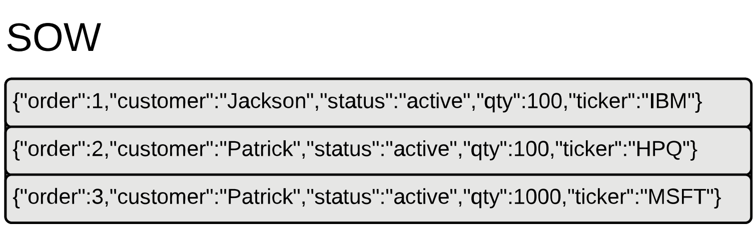
To be able to use delta messages, the message type for the subscription must support delta messages. All of the included AMPS message types, except for `binary`, support delta messages, with the limitations described in each section below. For custom message types, contact the message type implementer to determine whether delta support is implemented.

These features can often improve performance in environments where bandwidth is at a premium. Because these features require AMPS to parse, compare, and create messages, these features can consume somewhat more CPU on the AMPS server than a simple publish or subscribe, particularly for large messages with complex structure (such as deeply-nested XML).

11.1. Delta Subscribe

Delta subscribe allows applications to receive only the changed parts of a message when an update is made to a record in the SOW. When a delta subscription is active, AMPS compares the new state of the message to the old state of the message, creates a message for the difference, and sends the difference message to subscribers. Using this approach can simplify processing on the client side, and can improve performance when network bandwidth is the most important constraint.

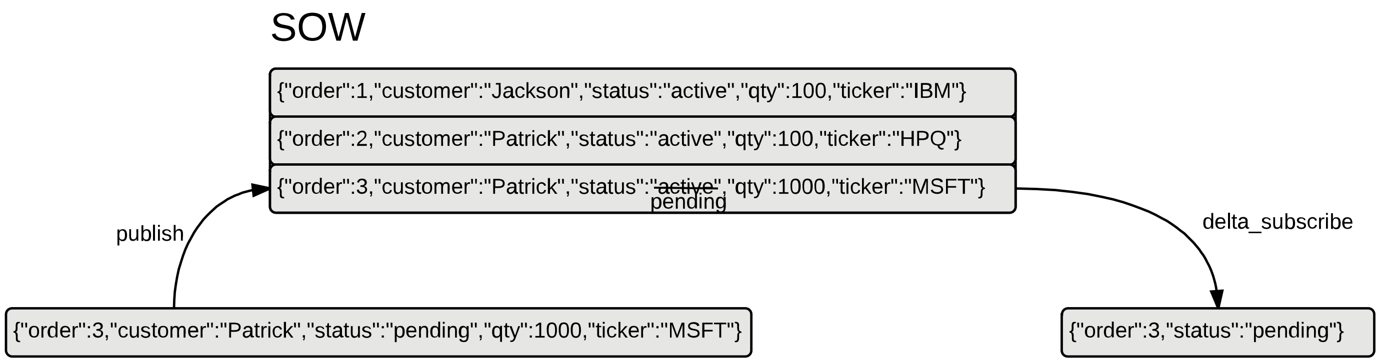
For example, consider a SOW that contains the following messages, with the `order` field as the key of the SOW topic:



Now, consider an update that changes the status of order number 3:

```
{"order":3,"customer":"Patrick","status":"pending","qty":1000,"ticker":"MSFT"}
```

For a regular subscription, subscribers receive the entire message. With a delta subscription, subscribers receive just the key of the SOW topic and any changed fields:



This can significantly reduce the amount of network traffic, and can also simplify processing for subscribers, since the only information sent is the information needed by the subscriber to take action on the message.

Using Delta Subscribe

Because a client must process delta subscriptions using substantially different logic than regular subscriptions, delta subscription is implemented as a separate set of AMPS commands rather than simply as an option on subscribe commands. AMPS supports two different ways to request a delta subscription:

Table 11.1. Delta subscribe commands

Command	Result
`delta_subscribe`	Register a delta subscription, starting with newly received messages.
`sow_and_delta_subscribe`	Replay the state of the SOW and atomically register a delta subscription.

Options for Delta Subscribe

The delta subscribe command accepts several options that control the precise behavior of delta messages:

Table 11.2. Options for delta subscribe

Option	Result
`no_empties`	Do not send messages if no data fields have been updated. By default, AMPS will publish a delta for every publish to the record, even if the data has not changed. By specifying this option, AMPS will only send messages when there is changed data.
`no_sowkeys`	Do not include the AMPS generated SowKey with messages. By default, AMPS includes this key to help you identify unique records within the SOW.
`send_keys`	Include the SOW key fields in the message. Because the SOW key fields indicate which message to update, without this option, updates to delta messages will never contain the SOW key fields. For views, the SOW key fields are the fields specified in the `Grouping` element.

Option	Result
	AMPS accepts this option for backward compatibility. As of AMPS 4.0, this option is included on delta subscriptions by default.
`oof`	AMPS will deliver out of focus messages on this subscription.

Identifying Changed Records

When an application that uses delta subscriptions receives a message, that message can either be a new record or an update to an existing record. AMPS offers two strategies for an application to tell whether the record is a new record or an existing record, and identify which record has changed if the message is an update to an existing record.

The two basic approaches are as follows:

1. By default, each message delivered through a delta subscription contains a SowKey header field. This field is the identifier that AMPS assigns to track a distinct record in the SOW. If the application has previously received a SowKey with that value, then the new message is an update to the record with that SowKey value. If the application has not previously received a SowKey with that value, then the new message contains a new record.

2. Delta messages can also contain the key fields from the SOW in the body of the message. This is controlled by the `send_keys` option on the subscription, which is always enabled as of AMPS 4.0. With this approach, the application parses the body of the message to find the key. If the application has previously received the key, then the message is an update to that existing record. Otherwise, the message contains a new record.

In either case, AMPS delivers the information the application needs to determine if the record is new or changed. The application chooses how to interpret that information, and what actions to take based on the changes to the record.

AMPS also supports out-of-focus notification for delta subscriptions, as described in Chapter 10. If your application needs to know when a record is deleted, expires, or no longer matches a subscription, you can use out-of-focus messages to be notified.

Delta Subscribe Support

To produce delta messages, the message type and the topic must both support delta subscribe. When this is not the case, AMPS accepts the subscription, but provides full messages rather than delta messages.

All of the basic message types provided with AMPS support delta subscribe with the exception of the `binary` message type. Composite message types support delta subscribe if they use the `composite-local` definition, as described in the section on composite message types.

AMPS queues do not support delta subscribe. AMPS accepts a delta subscription for a queue, but produces full messages from the queue.

All other AMPS topic types that are based on a SOW support delta subscribe. AMPS topics that do not use a SOW do not support delta subscribe, and instead produce full messages.

Multiple Subscriptions and Delta Subscribe

When a single connection to AMPS has multiple subscriptions, AMPS sends the message to that client once, with information on the set of subscriptions that match. AMPS sends a message that will include the requested data for all of the matching subscriptions. For example, if a message matches one subscription that requests full messages and another subscription from the same connection that requests deltas, both subscriptions will receive a full message. If your application depends on receiving deltas, take care that the application does not issue non-delta subscriptions for the same set of messages on the same connection.

11.2. Delta Publish

Delta publish allows publishers to update a message in the SOW by providing just the key fields for the SOW and the data to update. When AMPS receives a delta publish, AMPS parses the incoming message and the existing messages, identifies changed fields, and creates an updated message that merges changed fields from the publish into the existing message.

This can be particularly useful in cases where more than one worker acts on a record. For example, an order fullfillment application may need to check inventory, to ensure that the order is available, and check credit to be sure that the customer is approved for the order. These checks may be run in parallel, by different worker processes. With delta publish, each worker process updates the part of the record that the worker is responsible for, without affecting any other part of the record. Delta publish saves the worker from having to query the record and construct a full update, and eliminates the possibility of incorrect updates when two workers try to update the record at the same time.

For example, consider an order published to the SOW:

```
{"id":735,"customer":"Patrick","item":90123,"qty":1000,"state":"new"}
```

Using delta publishing, two independent workers can operate on the record in parallel, safely making updates and preparing the record for a final fulfillment process.

The inventory worker process is responsible for checking inventory. This worker subscribes to messages where the `/state = 'new' AND /inventory IS NULL AND /credit IS NULL`. This process receives the new message and verifies that the inventory system contains 1000 of the item # 90123. When it verifies this, it uses delta publish to publish the following update:

```
{"id":735,"inventory":"available"}
```

The credit worker process verifies that the customer is permitted to bill for the total amount. Like the inventory worker, this worker subscribes to messages where the `/state = 'new' and /inventory IS NULL and /credit IS NULL`. This process receives the new message and verifies that the customer is allowed to bill the total value of the order. When the check is complete, the credit worker publishes this message:

```
{"id":735,"credit":"approved"}
```

After both of these processes run, the SOW contains the following record:

```
{"id":735,"credit":"approved","inventory":"available",
 "customer":"Patrick","item":90123,"qty":1000,
 "state":"new"}
```

The fulfillment worker would subscribe to messages where `/state = 'new' AND /inventory IS NOT NULL AND /credit IS NOT NULL`.

Using Delta Publish

Because delta messages must be processed and merged into the existing SOW record, AMPS provides a distinct command for delta publish.

Table 11.3. Delta publish command

Command	Result
delta_publish	Publish a delta message. If no record exists in the SOW, add the message to the SOW. If a record exists in the SOW, merge the data from this record into the existing record.

Delta Publish Support

To accept delta publishes, the message type and the topic must both support delta publish. When this is not the case, AMPS accepts the publish, but may not produce the expected results.

All of the basic message types provided with AMPS support delta publish with the exception of the binary message type. Composite message types support delta publish if they use the composite-local definition, as described in the section on composite message types. The binary message, and types that do not support delta publish, produce the full, literal message provided with a delta publish command.

When a topic uses the composite-local message type, parts of the composite that are provided as empty (that is, zero-length) are considered to be unchanged, and the merged message contains the existing contents of that part. This provides a convenient way to update only one part of a composite message, without having to republish data that has not changed. For example, a composite-local type contains a JSON part and a binary part can modify the JSON part without having to republish the full binary part.

AMPS queues support delta publish to an underlying topic, if that underlying topic maintains a SOW. The merged message is provided to the AMPS queue.

All other AMPS topic types that are based on a SOW and accept publish commands support delta publish. AMPS topics that do not use a SOW do not support delta publish, so publishing a delta message to those topics produces the full, literal message from the publish command rather than a merged message. Without a SOW configured for the topic, AMPS does not track the current value of a message, and therefore does not have a way to merge the publish into an existing message.

Chapter 12. Message Acknowledgement

AMPS enables a client which sends commands to AMPS to request the status of those commands at various check points throughout the message processing sequence. These status updates are handled in the form of `ack` messages.

For many applications, it may not be necessary for the application to request message acknowledgements explicitly. The AMPS clients request a set of acknowledgements by default that balance performance with error detection.

AMPS supports a variety of `ack` types, and allows you to request multiple ack types on each command. For example, the received ack type requests that AMPS acknowledge when the command is received, while the completed ack type requests that AMPS acknowledge when it has completed the command (or the portion of the command that runs immediately). Each AMPS command supports a different set of types, and the precise meaning of the `ack` returned depends on the command that AMPS is acknowledging.

AMPS commands are inherently *asynchronous*, and AMPS does not provide acknowledgement messages by default. A client must both explicitly request an acknowledgement and then receive and process that acknowledgement to know the results of a command. It is normal for time to elapse between the request and the acknowledgement, and so AMPS acknowledgements provide ways to correlate the acknowledgement with the command that produced it. This is typically done with an identifier that the client assigns to a command, which is then returned in the acknowledgement for the command.

Acknowledgements for different commands may not arrive in the order that commands were submitted to AMPS. For example, a `publish` command to a topic that uses synchronous replication will not return a `persisted` acknowledgment until the synchronous replication destinations have persisted the message. If the client issues a `subscribe` command in the meantime, the `processed` acknowledgement for the `subscribe` command -- indicating that AMPS has processed the subscription request -- may well return before the `persisted` acknowledgement.

To see more information about the different commands and their supported acknowledgment types, please refer to the *AMPS Command Reference*, provided with 4.0 and greater versions of the AMPS clients and available on the 60East web site.

Chapter 13. Conflated Topics

To further reduce network bandwidth consumption, AMPS supports a feature called “conflated topics.” A conflated topic is a copy of one SOW topic into another with the ability to control the update interval.

To better see the value in a conflated topic, imagine a SOW topic called `ORDER_STATE` exists in an AMPS instance. `ORDER_STATE` messages are published frequently to the topic. Meanwhile, there are several subscribing clients that are watching updates to this topic and displaying the latest state in a GUI front-end.

If this GUI front-end only needs updates in five second intervals from the `ORDER_STATE` topic, then more frequent updates would be wasteful of network and client-side processing resources. To reduce network congestion, a conflated topic for the `ORDER_STATE` topic can be created which will contain a copy of `ORDER_STATE` updated in five second intervals. Only the changed records from `ORDER_STATE` will be copied to the conflated topic and then sent to the subscribing clients. Those records with multiple updates within the time interval will have their latest updated values copied to the conflated topic, and only those conflated values are sent to the clients. This results in substantial savings in bandwidth for records with high update rates. This can also result in substantial savings in processing overhead for a client.

13.1. SOW/ConflatedTopic

AMPS provides the ability to create ongoing snapshots of a SOW topic, called *conflated topics* (also called *topic replicas* in previous releases of AMPS). Topic replicas are updated on an interval, and store a snapshot of the current state of the world at each interval. This helps to manage bandwidth to clients that do not act on each update, such as a client UI that refreshes every second rather than with every update.

For compatibility with previous versions of AMPS, AMPS allows you to use `TopicReplica` as a synonym for `ConflatedTopic`.

Table 13.1. SOW/ConflatedTopic Parameters

Element	Description
`Name`	String used to define the name of the conflated topic. While AMPS doesn't enforce naming conventions, it can be convenient to name the conflated topic based on the underlying topic name. For example, if the underlying topic is `orders`, it can be convenient to name the conflated topic `orders-C`. If no `Name` is provided, AMPS accepts `Topic` as a synonym for `Name` to provide compatibility with versions of AMPS previous to 5.0.
`UnderlyingTopic`	String used to define the SOW topic which provides updates to the conflated topic. This must exactly match the name of a SOW topic.
`MessageType`	The message format of the underlying topic. This `MessageType` must be the `MessageType` of the provided `UnderlyingTopic`.
`Interval`	The frequency at which AMPS updates the data in the conflated topic. Default: `5 seconds`
`Filter`	Content filter that is applied to the underlying topic. Only messages that match the content filter are stored in the conflated topic.

```
<ConflatedTopic>
```

```
 <Topic>FastPublishTopic-C</Topic>
 <MessageType>nvfix</MessageType>
 <UnderlyingTopic>FastPublishTopic</UnderlyingTopic>
 <Interval>5s</Interval>
 <Filter>/region = 'A'</Filter>
</ConflatedTopic>
```

Chapter 14. Aggregating Data with View Topics

AMPS contains a high-performance aggregation engine, which can be used to project one SOW topic onto another, similar to the `CREATE VIEW` functionality found in most RDBMS software.

14.1. Understanding Views

Views allow you to aggregate messages from one or more SOW topics in AMPS and present the aggregation as a new SOW topic. AMPS stores the contents of the view in a user-configured file, similar to a materialized view in RDBMS software.

Views are often used to simplify subscriber implementation and can reduce the network traffic to subscribers. For example, if some clients will only process orders where the total cost of the order exceeds a certain value, you can both simplify subscriber code and reduce network traffic by creating a view that contains a calculated field for the total cost. Rather than receiving all messages and calculating the cost, subscribers can filter on the calculated field. You can also combine information from multiple topics. For example, you could create a view that contains orders from high-priority customers that exceed a certain dollar amount.

AMPS sends messages to view topics the same way that AMPS sends messages to SOW topics: when a message arrives that updates the value of a message in the view, AMPS sends a message on the view topic. Likewise, you can query a view the same way that you query a SOW topic.

Defining a view is straightforward. You set the name of the view, the SOW topic or topics from which messages originate and describe how you want to aggregate, or *project*, the messages. AMPS creates a topic and projects the messages as requested.

> ⚠ All message types that you specify in a view must support view creation. The AMPS default message types all support views.

Because AMPS uses the SOW topics of the underlying messages to determine when to update the view, the underlying topics used in a view must have a SOW configured. In addition, the topics must be defined in the AMPS configuration file before the view is defined.

14.2. Defining Views and Aggregations

Multiple topic aggregation creates a view using more than one topic as a data source. This allows you to enrich messages as they are processed by AMPS, to do aggregate calculations using information published to more than one topic. You can combine messages from multiple topics and use filtered subscriptions to determine which messages are of interest. For example, you can set up a topic that contains orders from high-priority customers.

You can join topics of different message types, and you can project messages of a different type than the underlying topic.

To create an aggregate using multiple topics, each topic needs to maintain a SOW. Since views maintain an underlying SOW, you can create views from views.

To define an aggregate, you decide:

- The topic, or topics, that contain the source for the aggregation
- If the aggregation uses more than one topic, how those topics relate to each other
- What messages to publish, or *project*, from the aggregation
- How to group messages for aggregation
- The message type of the aggregation

Message types provided with AMPS fully support views, with the following exceptions:

- `binary` message types cannot be an underlying topic for a view or the type of a view
- `protobuf` message types can be the underlying topic for a view, but cannot be the type of a view
- `composite-global` message types can be the underlying topic for the view, but cannot be the type of the view

If you are using a custom message type, check with the message type developer as to whether that message type supports aggregation.

Single Topic Aggregation: UnderlyingTopic

For aggregations based on a single topic, use the `UnderlyingTopic` element to tell AMPS which topic to use. All messages from the `UnderlyingTopic` will appear in the aggregation.

```
<UnderlyingTopic>MyOriginalTopic</UnderlyingTopic>
```

Multiple Topic Aggregation: Join

`Join` expressions tell AMPS how to relate underlying topics to each other. You use a separate `Join` element for each relationship in the view. Most often,the join expression describes a relationship between topics:

```
[topic].[field]=[topic].[field]
```

The topics specified must be previously defined in the AMPS configuration file. The square brackets `[]` are optional. If they are omitted, AMPS uses the first `/` in the expression as the start of the field definition. You can use any number of join expressions to define a multiple topic aggregation.

If your aggregation will join messages of different types, or produce messages of a different type than the underlying topics, you add message type specifiers to the join:

```
[messagetype].[topic].[field]=[messagetype].[topic].[field]
```

In this case, the square brackets `[]` around the *messagetype* are mandatory. AMPS creates a projection in the aggregation that combines the messages from each topic where the expression is true. In other words, for the expression:

```
<Join>[Orders].[/CustomerID]=[Addresses].[/CustomerID]</Join>
```

AMPS projects every message where the same `CustomerID` appears in both the `Addresses` topic and the `Orders` topic. If a `CustomerID` value appears in only the `Addresses` topic, AMPS does not create a projection for the message. If a `CustomerID` value appears in only the `Orders` topic, AMPS projects the message with `NULL` values for the `Addresses` topic. In database terms, this is equivalent to a `LEFT OUTER JOIN`.

You can use any number of `Join` expressions in an underlying topic:

```
<Join>[nvfix].[Orders].[/CustomerID]=[json].[Addresses].[/CustomerID]</Join>
<Join>[nvfix].[Orders].[/ItemID]=[nvfix].[Catalog].[/ItemID]</Join>
```

In this case, AMPS creates a projection that combines messages from the `Orders`, `Addresses`, and `Catalog` topics for any published message where matching messages are present in all three topics. Where there are no matching messages in the `Catalog` and `Addresses` topics, AMPS projects those values as `NULL`.

A `Join` element can also contain only one topic. In this case, all messages from that topic are included in the view.

Setting the Message Type

The `MessageType` element of the definition sets the type of the outgoing messages. The message type of the aggregation does not need to be the same as the message type of the topics used to create the aggregation. However, if the `MessageType` differs from the type of the topics used to produce the aggregation, you must explicitly specify the message type of the underlying topics.

For example, to produce JSON messages regardless of the types of the topics in the aggregation, you would use the following element:

```
<MessageType>json</MessageType>
```

Defining Projections

AMPS makes available all fields from matching messages in the join specification. You specify the fields that you want AMPS to project and how to project them.

To tell AMPS how to project a message, you specify each field to include in the projection. The specification provides a name for the projected field and one or more source field to use for the projected field. The data can be projected as-is, or aggregated using one of the AMPS aggregation functions, as described in Section 14.3 .

You refer to source fields using the XPath-like expression for the field. You name projected fields by creating an XPath-like expression for the new field. AMPS uses this expression to name the new field.

```
<Projection>
    <Field>[Orders].[/CustomerID]</Field>
    <Field>[Addresses].[/ShippingAddress] AS /DestinationAddress</Field>
    <Field>SUM([Orders].[/TotalPrice]) AS /AccountTotal</Field>
</Projection>
```

The sample above uses the `CustomerID` from the orders topic and the shipping address for that customer from the `Addresses` topic. The sample calculates the sum of all of the orders for that customer as the `AccountTotal`. The sample also renames the `ShippingAddress` field as `DestinationAddress` in the projected message.

Data Types and Projections

When projecting views, AMPS converts the original values into the AMPS internal type system and serializes those values into a new message. This approach allows AMPS to efficiently aggregate messages of different types and produce predictable results. The data type of the serialization is determined by the message type of the projected message: the message types provided by 60East in this release project the AMPS internal type.

This means that, for message types that rely on type markers to identify the type (such as `bson`), the type of the field in the projected message may reflect the AMPS internal type rather than the original type. This conversion is typically a widening conversion for numeric types (for example, input typed as a 32-bit integer will typically be widened to a 64-bit integer). For other types, the most common conversion is from a specific data type (such as regular expression) to a string type.

Grouping

Use grouping statements to tell AMPS how to aggregate data across messages and generate projected messages.

For example, an `Orders` topic that contains messages for incoming orders could be used to calculate aggregates for each customer, or aggregates for each symbol ordered. The grouping statement tells AMPS which way to group messages for aggregation.

```
<Grouping>
  <Field>[Orders].[/CustomerID]</Field>
</Grouping>
```

The sample above groups and aggregates the projected messages by `CustomerId`. Because this statement tells AMPS to group by CustomerId, AMPS projects a message for each distinct `CustomerId` value. A message to the Orders topic will create an outgoing message with data aggregated over the `CustomerId`.

Each field in the projection should either be an aggregate or be specified in the `Grouping` element. Otherwise, AMPS returns the last processed value for the field.

14.3. Functions

AMPS provides functions that you can use in your projections.

Aggregation Functions

These functions operate over groups of messages. They return a single value for each unique group.

Table 14.1. AMPS Aggregation Functions

Function	Description
AVG	Average over an expression. Returns the mean value of the values specified by the expression.
COUNT	Count of values in an expression. Returns the number of values specified by the expression.
MIN	Minimum value. Returns the minimum out of the values specified by the expression.

Function	Description
MAX	Maximum value. Returns the maximum out of the values specified by the expression.
SUM	Summation over an expression. Returns the total value of the values specified by the expression.

Null values are not included in aggregate expressions with AMPS, nor in ANSI SQL. COUNT will count only non-null values; SUM will add only non-null values; AVG will average only non-null values; and MIN and MAX ignore NULL values.

MIN and MAX can operate on either numbers or strings, or a combination of the two. AMPS compares values using the principles described for comparison operators. For MIN and MAX, determines order based on these rules:

- Numbers sort in numeric order.
- String values sort in ASCII order.
- When comparing a number to a string, convert the string to a number, and use a numeric comparison. If that is not successful, the value of the string is higher than the value of the number.

For example, given a field that has the following values across a set of messages:

```
24, 020, 'cat', 75, 1.3, 200, '75', '42'
```

MIN will return `1.3`, MAX will return `'cat'`. Notice that different message types may have different support for converting strings to numeric values: AMPS relies on the parsing done by the message type to determine the numeric value of a string.

14.4. Examples

Simple Aggregate View Example

For a potential usage scenario, imagine the topic `ORDERS` which includes the following NVFIX message schema:

Table 14.2. `ORDERS` Table Identifiers

NVFIX Tag	Description
OrderID	unique order identifier
Tick	symbol
ClientId	unique client identifier
Shares	currently executed shares for the chain of orders
Price	average price for the chain of orders

This topic includes information on the current state of executed orders, but may not include all the information we want updated in real-time. For example, we may want to monitor the total value of all orders executed by a client at any moment. If `ORDERS` was a SQL Table within an RDBMS, the “view” we would want to create would be similar to:

```
CREATE VIEW TOTAL_VALUE AS
```

```
SELECT ClientId, SUM(Shares * Price) AS TotalCost
FROM ORDERS
GROUP BY ClientId
```

As defined above, the `TOTAL_VALUE` view would only have two fields:

1. ClientId: the client identifier

2. TotalCost: the summation of current order values by client

Views in AMPS are specified in the AMPS configuration file in `View` sections, which are defined in the `SOW` section. The example above would be defined as:

```
<SOW>
  <Topic>
    <Name>ORDERS</Name>
    <MessageType>nvfix</MessageType>
    <Key>/OrderID</Key>
  </Topic>
  <View>
    <Name>TOTAL_VALUE</Name>
    <UnderlyingTopic>ORDERS</UnderlyingTopic>
    <FileName>./views/totalValue.view</FileName>
    <MessageType>nvfix</MessageType>
    <Projection>
      <Field>/ClientId</Field>
      <Field>SUM(/Shares * /Price) AS /TotalCost</Field>
    </Projection>
    <Grouping>
      <Field>/ClientId</Field>
    </Grouping>
  </View>
</SOW>
```

Views require an underlying SOW topic. See Chapter 7 for more information on creating and configuring SOW topics.

The `Topic` element is the name of the new topic that is being defined. This `Topic` value will be the topic that can be used by clients to subscribe for future updates or perform SOW queries against.

The `UnderlyingTopic` is the SOW topic or topics that the view operates on. That is, the `UnderlyingTopic` is where the view gets its data from. All XPath references within the `Projection` fields are references to values within this underlying SOW topic (unless they appear on the right-hand side of the `AS` keyword.)

The `Projection` section is a list of 1 or more `Field`s that define what the view will contain. The expressions can contain either a raw XPath value, as in “/ClientId” above, which is a straight copy of the value found in the underlying topic into the view topic using the same target XPath. If we had wanted to translate the `ClientId` tag into a different tag, such as `CID`, then we could have used the `AS` keyword to do the translation as in `/ClientId AS /CID`.

Unlike ANSI SQL, AMPS allows you to include fields in the projection that are not included in the `Grouping` or used within the aggregate functions. In this case, AMPS uses the last value processed for the value of these fields. AMPS enforces a consistent order of updates to ensure that the value of the field is consistent across recovery and restart.

An unexpected 0 (zero) in an aggregate field within a view usually means that the value is either zero or `NaN`. AMPS defaults to using 0 instead of `NaN`. However, any numeric aggregate function will result in a `NaN` if the aggregation includes a field that is not a number.

Finally, the `Grouping` section is a list of one or more `Field`s that define how the records in the underlying topic will be grouped to form the records in the view. In this example, we grouped by the tag holding the client identifier. However, we could have easily made this the “Symbol” tag `/Tick`.

In the below example, we group by the `/ClientId` because we want to count the number of orders *for each client* that have a value greater than 1,000,000:

```
<SOW>

  ...

  <View>
    <Name>NUMBER_OF_ORDERS_OVER_ONEMILL</Name>
    <UnderlyingTopic>ORDERS</UnderlyingTopic>
    <Projection>
      <Field>/ClientId</Field>
      <Field><![CDATA[SUM(IF(/Shares * /Price > 1000000, /Shares * /Price,
 NULL)) AS /AggregateValue]]> </Field>
      <Field>SUM(IF(/Shares * /Price &gt; 1000000, /Shares * /Price, NULL))
 AS /AggregateValue2</Field>
    </Projection>
    <Grouping>
      <Field>/ClientId</Field>
    </Grouping>
    <FileName>
      ./views/numOfOrdersOverOneMil.view
    </FileName>
    <MessageType>nvfix</MessageType>
  </View>

  ...

</SOW>
```

Notice that the `/AggregateValue` and `/AggregateValue_2` will contain the same value; however `/AggregateValue` was defined using an XML `CDATA` block, and `/AggregateValue_2` was defined using the XML `>` entity reference.

Since the AMPS configuration is XML, special characters in projection expressions must either be escaped with XML entity references or wrapped in a CDATA section.

Updates to underlying topics can potentially cause many more updates to downstream views, which can create stress on downstream clients subscribed to the view. If any underlying topic has frequent updates to the same records and/or a real-time view is not required, as in a GUI, then a replica of the topic may be a good solution to reduce the frequency of the updates and conserve bandwidth. For more on topic replicas, please see Chapter 13.

Multiple Topic Aggregate Example

This example demonstrates how to create an aggregate view that uses more than one topic as a data source. For a potential usage scenario, imagine that another publisher provides a COMPANIES topic which includes the following NVFIX message schema:

Table 14.3. COMPANIES Table Identifiers

NVFIX Tag	Description
CompanyId	unique identifier for the company
Tick	symbol
Name	company name

This topic includes the name of the company, and an identifier used for internal record keeping in the trading system. Using this information, we want to provide a running total of orders for that company, including the company name.

If ORDERS and COMPANIES were a SQL Table within an RDBMS, the “view” we would want to create would be similar to:

```
CREATE VIEW TOTAL_COMPANY_VOLUME AS
SELECT COMPANIES.CompanyId, COMPANIES.Tick, COMPANIES.Name,
 SUM(ORDERS.Shares) AS TotalVolume
FROM COMPANIES LEFT OUTER JOIN ORDERS
  ON COMPANIES.Tick = ORDERS.Tick
GROUP BY ORDERS.Tick
```

As defined above, the TOTAL_COMPANY_VOLUME table would have four columns:

1. CompanyId: the identifier for the company
2. Tick: The ticker symbol for the company
3. Name: The name of the company
4. TotalVolume: The total number of shares involved in orders

To create this view, use the following definition in the AMPS configuration file:

```
<SOW>
  <Topic>
    <Name>ORDERS</Name>
    <MessageType>nvfix</MessageType>
    <Key>/OrderID</Key>
    <FileName>./sow/%n.sow</FileName>
  </Topic>
  <Topic>
    <Name>COMPANIES</Name>
    <MessageType>nvfix</MessageType>
    <Key>/CompanyId</Key>
    <FileName>./sow/%n.sow</FileName>
  </Topic>
  <View>
    <Name>TOTAL_COMPANY_VOLUME</Name>
    <UnderlyingTopic>
```

```
      <Join>[ORDERS]./Tick = [COMPANIES]./Tick</Join>
    </UnderlyingTopic>
    <FileName>./views/totalVolume.view</FileName>
    <MessageType>nvfix</MessageType>
    <Projection>
      <Field>[COMPANIES]./CompanyId</Field>
      <Field>[COMPANIES]./Tick</Field>
      <Field>[COMPANIES]./Name</Field>
      <Field>SUM([ORDERS]./Shares) AS /TotalVolume</Field>
    </Projection>
    <Grouping>
      <Field>[ORDERS]./Tick</Field>
    </Grouping>
  </View>
</SOW>
```

As with the single topic example, first specify the underlying topics and ensure that they maintain a SOW database. Next, the view defines the underlying topic that is the source of the data. In this case, the underlying topic is a join between two topics in the instance. The definition next declares the file name where the view will be saved, and the message type of the projected messages. The message types that you join can be different types, and the projected messages can be a different type than the underlying message types. The projection uses three fields from the COMPANIES topic and one field that is aggregated from messages in the ORDERS topic. The projection groups results by the `Tick` symbols that appear in messages in the ORDERS topic.

Chapter 15. Transactional Messaging and Bookmark Subscriptions

AMPS includes support for transactional messaging, which includes persistence, consistency across restarts, and message replay. AMPS message queues use the transaction log to hold the messages in the queue. Transactional messaging is also the basis for replication, a key component of the high-availability capability in AMPS (as described in Chapter 25). AMPS message queues use the transaction log as a persistent record of the messages that have entered the queue, the order of those messages, and which messages have been acknowledged and removed from the queue. All of these capabilities rely on the AMPS *transaction log*. The transaction log maintains a record of messages. You can choose which messages are included in the transaction log by specifying the message types and topics you want to record.

The AMPS transaction log differs from transaction logging in a conventional relational database system. Unlike transaction logs that are intended solely to maintain the consistency of data in the system, the AMPS transaction log is fully queryable through the AMPS client APIs. For applications that need access to historical information, or applications that need to be able to recover state in the event of a client restart, the transaction log allows you to do this, relying on AMPS as the definitive single version of the state of the application. There is no need for complex logic to handle reconciliation or state restoration in the client. AMPS handles the difficult parts of this process, and the transaction log guarantees consistency.

Topics covered by a transaction log are able to provide reliable messaging with strict consistency guarantees.

When a transaction log is enabled, topics covered by the transaction log provide *atomic broadcast* from that instance. This means that the instance enforces a repeatable ordering on the messages, and guarantees that all subscribers receive messages reliably, in a consistent order, and with no gaps or duplicates.

15.1. Recording and Replaying Messages With Transaction Logs

AMPS includes the ability to record messages in a *transaction log*, and replay those messages at a later time. This capability is key for high availability, since it gives subscribers the ability to resume a subscription at a point in time without missing messages. This capability is also the foundation of replication, since it gives AMPS the ability to preserve message streams to be synchronized to an instance that has gone offline.

The transaction log in AMPS contains a sequential, historical record of messages. Each message is identified by a `bookmark`, a unique identifier that AMPS uses to locate the message within the overall set of recorded messages. The transaction log can record messages for a topic, a set of topics, or for filtered content on one or more topics.

An application can request a subscription that replays messages from the transaction log. Subscriptions that replay from the transaction log are called *bookmark subscriptions*, since the subscription begins at a specific point in the transaction log identified by a specific bookmark. Bookmark subscriptions provide topic and content filtering in the same way that normal subscriptions do, and provide a set of unique capabilities (such as the ability to pause and resume the subscription) that are made possible because the subscription is provided from a persistent record of the message stream. Bookmark subscriptions are also key to high availability with AMPS. When a client is recovering from a restart or failure, this ability to replay allows a client to fill gaps in received messages and resume subscriptions without missing a message. This feature also allows new clients to receive an exact replay of a message stream. Replay from the transaction log is also useful for auditing, quality assurance, and backtesting.

The transaction log is used in AMPS replication to ensure that all servers in a replication group are continually synchronized should one of them experience an interruption in service. For example, say an AMPS instance, as a member of a replication group, goes down. When it comes back up, it can query another AMPS instance for all of the messages it did not receive, thereby catching up to a point of synchronization with the other instances. This feature, when coupled with AMPS replication, ensures that message subscriptions are always available and up-to-date.

The AMPS transaction log records messages that are received from a publisher and events that affect those messages such as `sow_delete` commands. AMPS does not record messages that are created through a view, out-of-focus messages, or event status messages created by AMPS.

Understanding Message Persistence

To take advantage of transactional messaging, the publisher and the AMPS instance work together to ensure that messages are written to persistent storage. AMPS lets the publisher know when the message is persisted, so that the publisher knows that it no longer needs to track the message.

When a publisher publishes a message to AMPS, the publisher assigns each message a unique sequence number. Once the message has been written to persistent storage, AMPS uses the sequence number to acknowledge the message and let the publisher know that the message is persisted. Once AMPS has acknowledged the message, the publisher considers the message published. For safety, AMPS always writes a message to the local transaction log before acknowledging that the message is persisted. If the topic is configured for synchronous replication, all replication destinations have to persist the message before AMPS will acknowledge that the message is persisted.

For efficiency, AMPS may not acknowledge each individual message. Instead, AMPS acknowledges the most recent persisted message to indicate that all previous messages have also been persisted. Publishers that need transactional messaging do not wait for acknowledgment to publish more messages. Instead, publishers retain messages that haven't been acknowledged, and republish messages that haven't been acknowledged if failover occurs. The AMPS client libraries include this functionality for persistent messaging.

Configuring a Transaction Log

Before demonstrating the power of the transaction log, we will first show how to configure the transaction log in the AMPS configuration file.

```
❶<TransactionLog>
   ❷<JournalDirectory>./amps/journal/</JournalDirectory>
   ❸<JournalArchiveDirectory>
            /mnt/somedev0/amps/journal
   </JournalArchiveDirectory>
   ❹<PreallocatedJournalFiles>1</PreallocatedJournalFiles>
   ❺<MinJournalSize>10MB</MinJournalSize>
   ❻<Topic>
       <Name>orders</Name>
       <MessageType>nvfix</MessageType>
   </Topic>
   ❼<FlushInterval>40ms</FlushInterval>
</TransactionLog>
```

❶ All transaction log definitions are contained within the `TransactionLog` block. The following global settings apply to all `Topic` blocks defined within the `TransactionLog`: `JournalDirectory`, `PreallocatedJournalFiles`, and `MinJournalSize`.

❷ The JournalDirectory is the filesystem location where journal files and journal index files will be stored.
❸ The JournalArchiveDirectory is the filesystem location to which AMPS will archive journals. Notice that AMPS does not archive files by default. You configure an action to archive journal files, as described in Section 24.12.
❹ PreallocatedJournalFiles defines the number of journal files AMPS will create as part of the server startup. *Default: 2 Minimum: 1*
❺ The MinJournalSize is the smallest journal size that AMPS will create. *Default: 1GB Minimum: 10M*
❻ When a Topic is specified, then all messages which match exactly the specified topic or regular expression will be included in the transaction log. Otherwise, AMPS initializes the transaction logging, but does not record any messages to the transaction log.

The Topic section can be specified multiple times to allow for multiple topics to be published to the transaction log.
❼ The FlushInterval is the interval at which messages will be flushed the journal file during periods of slow activity. *Default: 100ms Maximum: 100ms Minimum: 30us*

Replaying Messages with Bookmark Subscription

One of the most useful and powerful features in AMPS is *bookmark subscription*, which is enabled by the transaction log. With bookmark subscription, an application requests a subscription that starts at a specific point in the transaction log. AMPS begins the subscription at the specified point, and provides messages from the transaction log.

Each message in the transaction log has a *bookmark*. A *bookmark* is an opaque, unique identifier that is added by AMPS to each message recorded in the transaction log. For messages provided from a transaction log, the field is included in the Bookmark header of the message. AMPS guarantees that bookmarks for the instance are monotonically increasing, which enables AMPS to rapidly find an individual bookmark within the transaction log.

A bookmark subscription simply requests that AMPS begin the subscription with the first message following the bookmark provided with the subscription. AMPS locates the bookmark in the transaction log, and begins the subscription at that point in time.

One way to think about a bookmark subscription is that AMPS publishes to the subscribing client only those messages that:

1. have bookmarks after the provided bookmark,
2. match the subscription's Topic and Filter, and
3. have been written to the transaction log

Because a bookmark subscription requires a transaction log, when a client requests a bookmark subscription for a topic that is not being recorded in the transaction log, AMPS returns an error.

AMPS allows an application to submit a comma-delimited list of bookmark values as the bookmark for a subscription request. In this case, AMPS begins replay at the oldest bookmark in the list. The client controls the bookmark provided on the subscription request. The AMPS server does not keep a persistent record of which bookmarks a specific client or subscription has processed. The AMPS client libraries provide a facilities for easily tracking the messages which an application has processed so the application can resume at the appropriate point in the transaction log.

Bookmark subscriptions are provided from the transaction log rather than the live publish stream. This lets AMPS adapt the pace of replay to the pace at which the subscriber is consuming replayed messages without triggering slow client offlining.

There are four different ways that a client can request a bookmark replay from the transaction log. Each of these bookmark types meets a different need and enables a different recovery strategy that an application can use. The sections below describe the recovery types, the cases in which they can be used, and how the 60East clients implement them.

> While there are similarities between a bookmark subscription used for replay and a SOW query, the transaction log and SOW are independent features that can be used separately. The SOW gives a snapshot of the current view of the latest data, while the journal is capable of playback of previous messages. Historical SOW queries provide a snapshot of the SOW at a defined point in the past, and are provided by the SOW database rather than the transaction log.

Recovery With an Epoch Bookmark

The epoch bookmark, when requested on a subscription, will replay the transaction log back to the subscribing client from the very beginning. Once the transaction log has been replayed in its entirety, then the subscriber will begin receiving messages on the live incoming stream of messages. A subscriber does this by requesting a `0` in the `bookmark` header field of their subscription. The AMPS clients provide a constant for epoch, typically represented as `EPOCH`.

This type of bookmark can be used in a case where the subscriber has begun after the start of an event, and needs to catch up on all of the messages that have been published to the topic.

To ensure that no messages from the subscription are lost during the replay, all message replay from the live subscription stream will be queued in AMPS until the client has consumed all of the messages from the replay stream. Once all of the messages from the replay stream have been consumed, AMPS will cut over to the live subscription stream and begin sending messages to the subscriber from that stream.

Bookmark Replay From NOW

The NOW bookmark, when requested on a subscription, declines to replay any messages from the transaction log, and instead begins streaming messages from the live stream - returning any messages that would be published to the transaction log that match the subscription's `Topic` and `Filter`.

This type of bookmark is used when a client is concerned with messages that will be published to the transaction log, but is unconcerned with replaying the historical messages in the transaction log. This strategy is often used for applications that want to ensure that they do not miss messages, even if the application temporarily loses connectivity, but are not concerned with older messages. For this case, the application subscribes with NOW when the application starts, and then re-establishes the subscription with the most recently-processed bookmark if connectivity is lost. This resubscription behavior is typically handled by the client reconnection logic (as in the 60East `HAClient` implementations).

The NOW bookmark is performed using a subscribe query with "0|1|" as the `bookmark` field. The AMPS clients provide a constant for this value, typically represented as `NOW`.

Bookmark Replay With a Bookmark

Clients that store the bookmarks from published messages can use those bookmarks to recover from an interruption in service. By placing a subscribe query with the last bookmark recorded, a client will get a replay of all messages persisted to the transaction log after that bookmark. Once the replay has completed, the subscription will then cut over to the live stream of messages.

To perform a bookmark replay, the client places a bookmark subscription with the bookmark at which to start the subscription.

Developer Note: the MOST_RECENT value

The AMPS client libraries provide a special constant value that requests that the library look up the bookmark for the most recently processed message in the bookmark store and then provide that bookmark in the subscription request. This special value is typically represented as `MOST_RECENT`. When the application requests a bookmark subscription with a bookmark of `MOST_RECENT`, the client library looks for the most recent bookmark processed by the application, then provides that bookmark for the subscription. This ensures that the subscription begins at last processed message, and the application receives the next unprocessed message for the subscription. If there is no record of a subscription, the AMPS clients will start with `EPOCH`.

It's important to remember that the AMPS server has no knowledge of the `MOST_RECENT` value. `MOST_RECENT` is never sent to AMPS and never appears in the AMPS log. `MOST_RECENT` is simply a request to the AMPS client library to look up the exact bookmark to provide to AMPS. The AMPS client libraries always translate a request for `MOST_RECENT` into either a specific bookmark value or `EPOCH`.

Bookmark Replay From a Moment in Time

The final type of bookmark supported is the ASCII-formatted timestamp. When using a timestamp as the bookmark value, the transaction log replays all messages that occurred after the timestamp, and then cuts over to the live subscription once the replay stream has been consumed.

This bookmark has the format of `YYYYmmddTHHMMSS[Z]` where:

- `YYYY` is the four digit year.
- `mm` is the two digit month.
- `dd` is the two digit day.
- `T` the character separator between the date and time.
- `HH` the two digit hour.
- `MM` the minutes of the time.
- `SS` the two digit second.
- `Z` is an optional timezone specifier. AMPS timestamps are always in UTC, regardless of whether the timezone is included. AMPS only accepts a literal value of `Z` for a timezone specifier.

For example, a timestamp for January 2nd, 2015, at 12:35:

```
20150102T123500Z
```

Content and Topic Filtering

As with all other subscriptions, bookmark subscriptions support content filtering.

Bookmark subscriptions provide only messages from topics that are recorded in the transaction log. In other words, when a bookmark subscription uses a topic regular expression, only messages from topics that are recorded in the transaction log are provided to the subscription. This ensures that a bookmark subscription provides a consistent, repeatable stream of messages. The topics provided to the subscription are the same during replay, when only messages recorded in the transaction log are available, and after replay completes, when every publish to AMPS is available. This also ensures that bookmark subscription that replays messages for a specific timeframe gets the same messages as bookmark subscribers that had active subscriptions during that timeframe.

Content filtering is covered in greater detail in Chapter 5.

Delivery Rate Control for Bookmark Subscriptions

AMPS allows subscribers to specify the maximum delivery rate for messages delivered from a bookmark subscription. A subscriber specifies the maximum rate at which AMPS should deliver messages to the subscription. AMPS then limits the rate at which replay occurs so that the overall rate does not exceed the specified maximum. Rate control is not available for subscriptions that use the `live` option.

To request rate control, a subscriber provides the `rate` option on the subscription. A rate can be specified in either messages per second or number of bytes delivered per second. For example, the following subscription option limits delivery to 1000 messages per second:

```
rate=1000
```

To limit delivery to 500KB per second, a subscriber would provide this option:

```
rate=500KB
```

.

Pausing and Resuming Bookmark Subscriptions

Beginning in AMPS 5.0, AMPS offers the ability to pause a bookmark subscription. When a subscriber requests that AMPS pause the subscription, AMPS stops providing messages from the bookmark subscription, but does not remove the subscription. The subscriber can then resume the subscription, and AMPS will again begin providing messages from the subscription. While the subscription is paused, AMPS maintains a record of the current position in the transaction log, and begins replay from that point.

This feature can be useful for clients that need to temporarily stop processing messages while minimizing the buffer space consumed during the time that the client is not consuming messages. For example, a simulation that visualizes historical data might pause the bookmark subscription if the user pauses the visualization.

An application may create a subscription in the paused state by including `pause` as an option on the initial `subscribe` command. To pause an active subscription, a subscriber sends a `subscribe` command with the existing subscription ID and the `pause` option. To resume a subscription, a subscriber sends a `subscribe` command with the subscription ID (or a comma-separated list of subscription IDs) and the `resume` option. The AMPS clients provide convenience constants for the `pause` and `resume` options.

When multiple bookmark subscriptions are resumed at the same time, AMPS will attempt to combine replay for the subscriptions. When AMPS can combine replay, AMPS will guarantee that messages across subscriptions are delivered from the same replay, which can help to preserve order across subscriptions. AMPS can combine subscriptions when they are delivered to the same client connection, were paused at the same bookmark, deliver at the same rate and are resumed with the same command. This feature can be useful for synchronizing message delivery across a number of subscriptions. When using `pause` and `resume` for this purpose, an application typically includes the `pause` option on a number of subscriptions when the subscriptions are created, and then resumes the subscriptions when the application is ready to begin the replay.

Pausing a subscription stops AMPS from sending messages to the client once the pause command is processed. However, any messages already on the network, or in a network buffer on the client or the server will be delivered to the client.

AMPS allows you to begin a subscription in the paused state by providing the `pause` option when creating the subscription.

AMPS removes a paused subscription if the subscriber disconnects: for restarting a subscription across subscriber restarts, use the basic bookmark subscription features as described above.

Using the 'live' Option for Bookmark Subscriptions

Once replay from the transaction log is finished, AMPS sends messages to subscribers as the messages are processed. By default, AMPS waits until a message is persisted to the transaction log before sending the message to subscribers. Because each message delivered is persisted, this approach ensures that the sequence of messages is consistent across client and server restarts, and that no messages will be missed or duplicated during failover.

In some cases, reducing latency may be more important than consistency. To support these cases, AMPS provides a `live` option on bookmark subscriptions. For bookmark subscriptions that use the `live` option, once replay has finished, AMPS sends messages to subscribers *before* the message has been persisted. This can provide a small reduction in latency at the expense of increasing the risk of inconsistency upon failover. For example, if a publisher does not republish a message after failover, your application may receive a message that is not stored in the transaction log and that other applications have not received.

The `live` option increases the risk of inconsistent data between your program and AMPS in the event of a failover. 60East recommends using this option only if the risk is acceptable and if your application requires the small latency reduction this option provides.

Because the `live` option does not wait for messages to be persisted, subscriptions that use this option are subject to slow client offlining after replay from the transaction log is complete.

The `rate`, `pause`, and `resume` options are not supported with the `live` option.

Managing Journal Files

The design of the journal files for the transaction log are such that AMPS can archive, compress and remove these files while AMPS is running. AMPS actions provide integrated adminstration for journal files, as described in Chapter 24.

Archiving a file copies the file to an archival directory, typically located on higher-capacity but higher-latency storage. Compressing a file compresses the file in place. Archived and compressed journal files are still accessible to clients for replay and for AMPS to use in rebuilding any SOW files that are damaged or removed.

When defining a policy for archiving, compressing or removing files, keep in mind the amount of time for which clients will need to replay data. Once journal files have been deleted, the messages in those files are no longer available for clients to replay or for AMPS to use in recreating a SOW file. If journal files are removed, and a SOW file is retained, this means that the SOW may have data that is not in the transaction log.

To determine how best to manage your journal files, consider your application's access pattern to the recorded messages. Most applications have a period of time (often a day or a week) where historical data is in heavy use, and a period of time (often a week, or a month) where data is infrequently used. One common strategy is to create the journal files on high-throughput storage. The files are archived to slower, higher-capacity storage after a short period of time, compressed, and then to removed after a longer period of time. This strategy preserves space on high-throughput storage, while still allowing the journals to be used. For example, if your applications frequently replay data for the last day, occasionally replay data older than the last week, and never request data older than one month, a management strategy that meets these needs would be to archive files after one day, compress them after a week, and remove them after one month.

> ⚠ If you remove journal files when AMPS is shut down, keep in mind that the removal of journal files must be sequential and can *not* leave gaps in the remaining files. For example, say there are three journal files, `001`, `002` and `003`. If only `002` is removed, then the next AMPS restart could potentially overwrite the journal file `003`, causing an unrecoverable problem.

When using AMPS actions to manage journal files, AMPS ensures that all replays from a journal file are complete and all messages from a journal file have been successfully replicated before removing the file.

Chapter 16. Message Queues

AMPS includes high performance queuing built on the AMPS messaging engine and transaction log. AMPS message queues combine elements of classic message queuing with the advanced messaging features of AMPS, including content filtering, aggregation and projection, historical replay, and so on. This chapter presents an overview of queues.

AMPS message queues help you easily solve some common messaging problems:

- Ensuring that a message is only processed once
- Distributing tasks across workers in a fair manner
- Ensuring that a message is delivered to and processed by a worker
- Ensuring that when a worker fails to process a message, that message is redelivered

While it's possible to create applications with these properties by using the other features of AMPS, message queues provide these functions built into the AMPS server. In addition, message queues allow you to:

- Replicate messages between AMPS instances while preserving delivery guarantees
- Create views and aggregates based on the current contents of a queue
- Filter messages into and out of a queue
- Provide a single published message to multiple queues
- Aggregate multiple topics into a single queue

Use message queues when you need to ensure that a message is processed by a single consumer. When you need to distribute messages to a number of consumers, use the AMPS pub/sub delivery model.

16.1. Getting Started with AMPS Queues

To add a simple queue to AMPS, add the following options to your configuration file.

First, create a transaction log that will record the messages for the queue, as described in Chapter 15. You add the transaction log entry if your AMPS configuration does not already have one. Otherwise, you can simply add a `Topic` statement or modify an existing `Topic` statement to record the messages. The sample below captures any JSON messages published to topics that end with `_Queue`.

```
<AMPSConfig>
    ...
    <TransactionLog>
         <JournalDirectory>./journals</JournalDirectory>
         <Topic>
            <Name>.*_Queue$</Name>
            <MessageType>json</MessageType>
         </Topic>
```

```
    </TransactionLog>
   ...
</AMPSConfig>
```

Next, declare the queue topic itself. Queues are defined in the SOW element of the `AMPSConfig` file, as shown below:

```
<AMPSConfig>
   ...

   <SOW>
       <Queue>
           <Name>Simple_Queue</Name>
           <MessageType>json</MessageType>
           <Semantics>at-most-once</Semantics>
       </Queue>
   </SOW>

   ...
</AMPSConfig>
```

These simple configuration changes create an AMPS message queue. Notice that the `Topic` for the queue in this case is `Simple_Queue`, which matches the regular expression configured for the transaction log.

This simple queue provides each message that arrives for the queue to at most one subscriber. After AMPS delivers the message to one subscriber, AMPS removes the message from the queue without waiting for the subscriber to acknowledge the message.

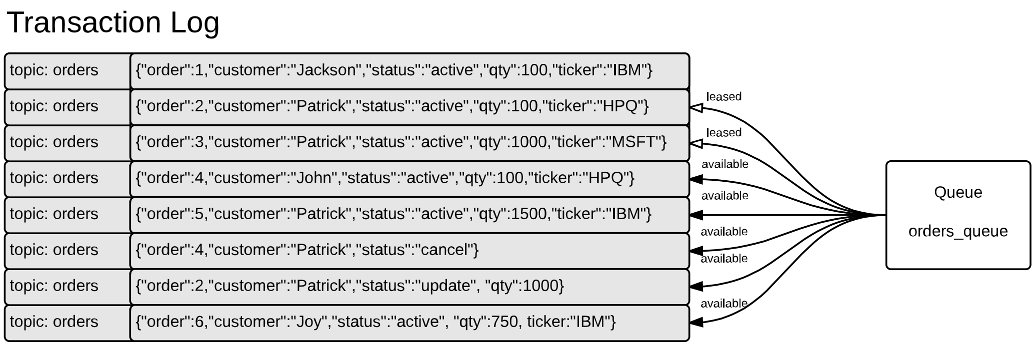
While it's easy to create a simple queue, AMPS offers a rich queuing model that is designed to meet a wide variety of queueing needs. The options are described in the following sections and the *AMPS Configuration Guide*.

16.2. Understanding AMPS Queuing

AMPS message queues take advantage of the full historical and transactional power of the AMPS engine. Each queue is implemented as a view over an underlying topic or set of topics. Each of the underlying topics must be recorded in a transaction log. Publishers publish to the underlying topic, and the messages are recorded in the transaction log. Consumers simply subscribe to the queue. AMPS tracks which messages have been delivered to subscribers and which messages have been processed by subscribers. AMPS delivers the next available message to the next subscriber.

Unlike traditional queues, which require consumers to poll for messages, AMPS queues use a subscription model. In this model, each queue consumer requests that AMPS provide messages from the queue. The consumer can also request a maximum number of messages to have outstanding from the queue at any given time, referred to as the *backlog* for that consumer. When a message is available, and the consumer has fewer messages outstanding than the backlog for that consumer, AMPS delivers the message to the consumer. This improves latency and conserves bandwidth, since there is no need for consumers to repeatedly poll the queue to see if work is available. In addition, the server maintains an overall view of the consumers, which allows the server to control message distribution strategies to optimize for latency, optimize to prefer delivery to clients with the most unused capacity, or optimize for general fairness.

The following diagram presents a simplified view of an AMPS queue.



As the diagram indicates, a queue tracks a set of messages in the transaction log. The messages the queue is currently tracking are considered to be in the queue. When the queue delivers a message, it marks the message as having been delivered (shown as *leased* in the diagram above). Messages that have been processed are no longer tracked by the queue (for example, the message for the order 1 in the diagram above). When a message has been delivered and processed, that event is recorded in the transaction log to ensure that the queue meets the delivery guarantees even across restarts of AMPS.

Because queues are implemented as views over underlying topics, AMPS allows you to create any number of queues over the same underlying topic. Each queue tracks messages to the topic independently, and can have different policies for delivery and fairness. When a queue topic has a different name than the underlying topic, you can subscribe to the underlying topic directly, and that subscription is to the underlying (non-queue) topic. When a queue topic has the same name as the underlying topic (the default), all subscriptions to that topic are to the queue.

Likewise, AMPS queues work seamlessly with the AMPS entitlement system. Permissions to queues are managed the same way permissions are managed to any other topic, as described in the Entitlements section of the *AMPS User Guide*.

AMPS queues provide a variety of options to help you tailor the behavior of each queue to meet your application's needs.

Delivery Semantics

AMPS supports two different levels of guarantees for queue delivery:

- With `at-least-once` delivery, AMPS delivers the message to one subscriber at a time, and expects that subscriber to explicitly remove the message from the queue when the message has been received and processed. With this guarantee, each message from the queue must be processed within a specified timeout, or *lease period*. AMPS tracks the amount of time since the message was sent to the subscriber. If the subscriber has not responded by removing the message within the lease period, AMPS revokes the lease and the message is available to another subscriber.

  In this model, receiving a message is the equivalent of a non-destructive get from a traditional queue. To acknowledge and remove the message, a subscriber uses the `sow_delete` command with the bookmark of the message.

  Leases are broken and messages are returned to the queue if the lease holder disconnects from AMPS. This ensures that, if a message processor fails or loses its connection to AMPS, the message can immediately be processed by another message processor.

- With `at-most-once` delivery, AMPS removes the message from the queue as soon as the message is sent to a subscriber. However, the subscriber still needs to acknowledge that the message was processed, so that AMPS can track the subscription backlog, as described below.

In this model, receiving a message is the equivalent of a destructive get from a traditional queue. The message is immediately removed by AMPS, and is no longer available in the queue.

Subscription Backlog

For efficiency, queues in AMPS use a push model of delivery, providing messages to consumers when the message becomes available rather than requiring the consumer to poll the queue. To manage the workload among consumers, AMPS queues keep track of a *subscription backlog*. This backlog is the number of messages that have been provided to an individual subscription that have not yet been acknowledged. This backlog helps AMPS provide strong delivery guarantees while still optimizing for high throughput processing. AMPS calculates the subscription backlog for each subscription by calculating:

- The *minimum* `MaxPerSubscriptionBacklog` setting for the queues matched by the subscription, *or*
- The `max_backlog` specified on the subscribe command,

whichever is *smallest*

Notice that, if a subscriber does not provide a `max_backlog` on a subscription, AMPS defaults to a `max_backlog` of `1`. In practical terms, this means that an application must explicitly specify a backlog to be able to receive more than one message from a queue at a time, regardless of the queue configuration.

Subscribers request a `max_backlog` by adding the request to the options string of the `subscribe` command. For example, to request a `max_backlog` of 10, a subscriber includes `max_backlog=10` in the options for the command.

To improve concurrency for subscribers, 60East recommends using a backlog of at least `2`. This allows efficient pipelined delivery, as the consumer can be processing one message while the previous message is being acknowledged. With a `max_backlog` higher than `1`, the consumer never needs to be stopped waiting for the next message from the queue.

Delivery Fairness

When a queue provides `at-least-once` delivery, AMPS provides three different algorithms for distributing messages among subscribers. Each algorithm has different performance and fairness guarantees. For `at-most-once` delivery, AMPS supports only the `round-robin` method of distributing messages.

Table 16.1. Message Distribution Algorithms

Algorithm	Description
`fast`	This strategy optimizes for the lowest latency. AMPS delivers the message to the first subscription found that does not have a full backlog. With this algorithm, AMPS tries to minimize the time spent determining which subscription receives the message without attempting to distribute messages fairly across subscriptions.
`round-robin`	This strategy optimizes for general fairness across subscriptions. AMPS delivers the message to the next available subscription that does not have a full backlog. With this algorithm, AMPS delivers messages evenly among the subscribers that have space in their backlog.
`proportional`	This strategy optimizes for delivery to subscriptions with the most unused capacity. AMPS delivers the message to the subscription that has the highest proportion of

Algorithm	Description
	backlog capacity unused. AMPS determines this by taking the ratio of unacknowledged messages to the maximum backlog. For example, if there are three active subscribers for the queue, with backlog settings and outstanding messages as follows: **Table 16.2. Proportional delivery example** Subscriber / Unacknowledged Messages / Maximum Backlog Inky / 1 / 2 Blinky / 3 / 4 Clyde / 4 / 10 In this case, with `proportional` delivery, a new message for the queue will be delivered to Clyde, since that subscriber has only filled 40% of the backlog, as compared with 50% for Inky and 75% for Blinky.

Table 16.2. Proportional delivery example

Subscriber	Unacknowledged Messages	Maximum Backlog
Inky	1	2
Blinky	3	4
Clyde	4	10

AMPS defaults to `proportional` delivery for `at-least-once` queues and defaults to `round-robin` (the only valid delivery model) for `at-most-once` queues.

Acknowledging Messages

Subscribers must acknowledge each message to indicate to AMPS that a message has been processed. The point at which a subscriber acknowledges a message depends on the exact processing that the subscriber performs and the processing guarantees for the application. In general, applications acknowledge messages at the point at which the processing has a result that is durable and which would require an explicit action (such as another message) to change. Some common points at which to acknowledge a message are:

- When processing is fully completed
- When work is performed that would require a compensating action (that is, when information is committed to a database or forwarded to a downstream system)
- When work is submitted to a processor that is guaranteed to either succeed or explicitly indicate failure

To acknowledge a message, the subscriber uses the acknowledge convenience methods in the AMPS client. These commands issue a `sow_delete` command with the bookmark from the message to acknowledge. AMPS allows subscribers to acknowledge multiple messages simultaneously by providing a comma-delimited list of bookmarks in the `sow_delete` command: the AMPS clients provide facilities for batching acknowledgements for efficiency.

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

Message Flow for Queues

The message flow for AMPS queues is as follows. The message flow differs depending on whether the queue is configured for `at-most-once` delivery or `at-least-once` delivery.

When the queue is configured for `at-most-once` delivery:

1. A publisher publishes a message to an underlying topic.
2. The message becomes available in the queue.
3. The message is published to a subscriber when:
   - There is a subscription that matches the message, and the subscriber is entitled to see the message
   - The message is the oldest message in the queue that matches the subscription
   - The subscription has remaining capacity in its backlog
   - The subscription is the next subscription to receive the message as determined by the delivery fairness for the queue

   AMPS removes the message from the queue when the message is published.
4. If no subscription has requested the message, and the message has been in the queue longer than the `Expiration` time, AMPS removes the message from the queue. With AMPS queues, message expiration is considered to be a normal way for the message to leave the queue, and is not considered an error.
5. The subscriber processes the message, and acknowledges the message when processing is finished to indicate to AMPS that the subscriber has capacity for another message.

When the queue is configured for `at-least-once` delivery:

1. A publisher publishes a message to an underlying topic.
2. The message becomes available in the queue.
3. The message is published to a subscriber when:
   - There is a subscription that matches the message
   - The message is the oldest unleased message in the queue that matches the subscription
   - The subscription has remaining capacity in its backlog
   - The subscription is the next subscription to receive a message as determined by the delivery fairness for the queue

   AMPS calculates the lease time for the message and provides that time to the subscriber along with the message.
4. If the message has been in the queue longer than the `Expiration` time, and there is no current lease on the message, AMPS removes the message from the queue.
5. If a subscriber has received the message, but has not removed the message from the queue at the time the lease expires, AMPS returns the message to the queue if the message has been in the queue less than the `Expiration` time. If the message has been in the queue longer than the `Expiration` time, AMPS removes the message from the queue when the lease expires.
6. The subscriber processes the message, and removes the message from the queue by acknowledging the message (which is translated by the client into the appropriate `sow_delete` command).

Advanced Messaging and Queues

Queues are implemented as AMPS topics which lets you use the advanced messaging features of AMPS to create your queues and provide insight into your queues. For example, consumers can use content filtering to select the messages from the queue that they want to consume. You can use content filters to select only a subset of messages published to an underlying topic to populate the queue. You can even create a view that aggregates data from multiple topics, and use that view as the underlying topic for the queue. Since messages for queues are recorded in the transaction log, you can easily replay messages published from the queue using a bookmark subscription.

SOW and Queues

AMPS fully supports SOW topics as the underlying topic for a queue. Each publish to the SOW is considered to be an update to the SOW record, and creates a new message in the queue.

AMPS allows you to run a SOW query over a queue. The query returns the current available messages from the queue.

AMPS does not provide Out-of-Focus messages for subscriptions to queues. Each change to the underlying SOW creates a new message in the queue, rather than generating an OOF message.

Delta Messaging with Queues

AMPS delta subscriptions rely on being able to determine the last state of a message delivered on a subscription and providing a set of changes to the subscriber. With AMPS queues, AMPS treats each update to a SOW record as a new message, so there's no previous state that would generate a delta message. When the underlying topics of a queue are SOW topics, AMPS supports delta publish to those topics.

AMPS allows delta subscriptions to a queue, but treats each message as a new publish and delivers the full message.

Views over Queues

AMPS fully supports creating a view with a queue as an underlying topic. The view aggregates the messages that are currently available in the queue. When a message is leased, that message is no longer available to the queue and does not appear in the view. If the message is returned to the queue, then the message is again available to the view and appears in the view. When a message expires, that message is no longer available in the view.

Views over queues can be useful to show constantly-updated aggregates of the activity in the queue. For example, you could create an aggregate that shows the total value of unprocessed orders currently in the queue.

Bookmark Subscriptions and Queues

The queue itself does not provide redelivery or replay of messages. Therefore, AMPS translates a bookmark subscription to a queue to be a bookmark subscription to the underlying topic for the queue. This allows you to replay messages from the underlying topic *without* queue delivery semantics. A bookmark subscription to a queue becomes a publish/subscribe bookmark subscription to the underlying topic. Messages from this subscription do not have `at-most-once` or `at-least-once` delivery, and do not need to be acknowledged. The subscription is a publish/subscribe bookmark subscription, just as though there was no queue for the topic.

To get queueing semantics, do not include a bookmark on subscriptions to a queue.

16.3. Replacing Queue Subscriptions

Queues support the `replace` option for subscriptions. As with subscriptions to other topics, queue subscriptions can replace the content filter, the topic, the options, or all of the above. Replacement is atomic. The queue consumer is guaranteed that, after the replace occurs, only messages that match the new subscription will be delivered.

Replacing queue subscriptions differs from unsubscribing and resubscribing with new parameters in two ways:

1. AMPS does not break message leases or adjust the number of currently-unacknowledged messages for the subscription, even if the messages no longer match the current subscription. AMPS makes no assumptions about the state of the messages, and requires the subscriber to acknowledge them or allow the lease to expire.

2. AMPS may change the maximum backlog for the subscription if either the `max_backlog` option the topic for the subscription has changed. AMPS adjusts the backlog using the same logic as when the subscription was entered: the maximum backlog will be the smaller of the option set by the consumer or the limit on the queue. This can result in a situation where the consumer has more messages leased than the current maximum for the subscription, and no new messages will be delivered until that number drops below the current maximum.

   For example, if the consumer has a requested max_backlog of 10 and updates a subscription from a queue with a configured maximum of 10 to a queue with a configured maximum of 5, the new backlog for the subscription will be 5. However, the consumer may still have 10 messages outstanding.

In all other ways, AMPS behaves as though the replaced subscription was a new subscription to the queue.

16.4. SOW/Queue

This section lists configuration parameters for queues.

The `Queue` tag is used to configure message queues. AMPS accepts `QueueDefinition` as a synonym for `Queue`.

Table 16.3. Queue configuration elements

Element	Description
`Name`	The name of the queue topic. This name is the name that consumers subscribe to. If no `Name` is provided, AMPS accepts `Topic` as a synonym for `Name` in the `Queue` definition.
`MessageType`	The message type of the queue.
`UnderlyingTopic`	A topic name or regular expression for the topic that contains the messages to capture in the queue. These topics must be recorded in a transaction log, and all must be of the same message type as the queue. If an `UnderlyingTopic` is not provided, the `UnderlyingTopic` defaults to the Topic.
`DefaultPublishTarget`	The topic to publish to when an application publishes a message to the queue. For simplicity, AMPS allows applications to publish messages to the queue, and for those messages to be routed to one of the underlying topics. This element is required if the `UnderlyingTopic` contains regular expression characters. Otherwise, the `UnderlyingTopic` is a single topic and this element is optional and defaults to the `UnderlyingTopic`.
`LeasePeriod`	The amount of time that a subscriber has ownership of the message before the message is returned to the queue. For `at-least-once` delivery semantics, the consumer must

Element	Description
	process and acknowledge the message within this lease period, or the message may be provided to another subscriber. The `LeasePeriod` is measured from the time that AMPS sends the message to the subscriber. Set the `LeasePeriod` to account for round trip network latency as well as the expected processing time for the subscribers. Default: `infinite` (no expiration)
`Semantics`	The delivery semantics to use for this queue. There are two accepted values: • `at-least-once` With these semantics, you can guarantee that a message has been processed by at least one subscriber, as described in the introduction to Queues in the *AMPS User Guide*. With this value, a subscriber must explicitly remove the message from the queue once the message is processed. • `at-most-once` With these semantics, AMPS removes the message from the queue immediately when AMPS sends the message. This allows you to guarantee that no more than one subscriber will process the message. Default: `at-least-once`
`MaxBacklog`	The maximum number of outstanding, unacknowledged messages in the queue at any one time. This parameter allows you to set limits on the number of pending messages from the queue overall. When the queue reaches the MaxBacklog, no incoming messages are delivered from the queue until a message is removed from the queue (either by expiring, or being acknowledged by a client). This parameter allows you to avoid overwhelming clients during periods of heavy activity. Notice that this does not set a limit of any sort on the capacity of the queue. This parameter allows you to limit the number of messages that the queue will make available to subscribers at a given time, but does not restrict the capacity of the queue to track messages. Default: `infinite`
`MaxPerSubscriptionBacklog`	The maximum number of outstanding, unacknowledged messages in the queue for an individual subscription. This parameter allows you to avoid overwhelming a single subscriber during a period of heavy activity. Subscribers can declare the maximum number of messages that the subscription is prepared to lease at a given time. This maximum defaults to 1 when there is no maximum explicitly specified for a subscription. AMPS will lease the

Element	Description
	number specified in the subscription or the maximum set for the queue, whichever is lower. Notice that this does not set a limit of any sort on the capacity of the queue. This parameter allows you to limit the number of messages that the queue will make available for a single subscription at a given time, but does not restrict the capacity of the queue to track messages. Default: `1`
`Expiration`	The length of time a message can remain in the queue before AMPS considers the message undeliverable. Messages may expire while a subscriber has a lease on the message. AMPS does not send an additional notification in this case. Default: `infinite`
`Filter`	An AMPS `Filter` that is applied to the `UnderlyingTopic`. When a `Filter` is specified, only messages matching the `Filter` appear in the queue. By default, there is no filter and all messages from the UnderlyingTopic are presented in the queue.
`RecoveryPoint`	This option allows you to specify the point at which AMPS begins reviewing the transaction log to recover the state of the queue when AMPS restarts. By default, AMPS reviews the full log to determine the contents and state of the queue. The `RecoveryPoint` can be one of the following: • `epoch` - Recovery begins at the beginning of the transaction log • `creation` - Recovery begins at the time the queue was created • AMPS bookmark - When an AMPS bookmark is provided, AMPS starts recovery at the specified bookmark. • ISO-8601 timestamp - When a timestamp is provided, AMPS starts recovery at the specified timestamp. Default: `epoch`
`FairnessModel`	AMPS provides different methods to distribute messages across active subscriptions: • `fast` - AMPS delivers to the first subscription found that can process the message • `round-robin` - AMPS distributes to the next subscription found that can process the message

Element	Description
	• `proportional` - AMPS delivers to the subscription with the lowest ratio of active messages to available backlog Default: `proportional` for `at-least-once` queues, `round-robin` for `at-most-once` queues
`Leasing`	Ownership model for leased messages. AMPS supports the following models: • `strict` - AMPS allows a client to acknowledge (`sow_delete`) only messages that are leased to the client or currently unleased. If a client acknowledges a message leased to another client, there is no effect. • `sublet` - AMPS allows any client to acknowledge any message, regardless of whether another client has a lease on the message. Default: `sublet`

```
<!--
    Notice that the topics to use for
    the queue (ORDERS_.*) must be
    recorded in a transaction log.
-->

<SOW>
  <Queue>
    <Name>MQ</Name>
    <MessageType>json</MessageType>
    <UnderlyingTopic>ORDERS_.*</UnderlyingTopic>
    <DefaultPublishTarget>ORDERS_DIRECT</DefaultPublishTarget>
    <LeasePeriod>60s</LeasePeriod>
    <Expiration>1d</Expiration>
    <MaxBacklog>3</MaxBacklog>
  </Queue>
</SOW>
```

Example 16.1. Queue Example

Chapter 17. Message Types

Message communication between the publisher and subscriber in AMPS is managed through the use of message types. Message types define the data contained within an AMPS message. Each topic has a specific message type. Transports used for publishers and subscribers can also define specific message types. For a given transport, AMPS only process messages of the type or types that the transport accepts.

When AMPS needs to use the data within a message, AMPS uses the message type to parse the message into an internal representation. AMPS uses the same internal representation for all message types. Likewise, if AMPS needs to create a new message from a set of values (for example, for a view), AMPS uses the message type to serialize that set of values into the correct format. AMPS filters, commands, processing flow, and so forth are the same for every message type. Message types do not change how AMPS processes messages. A message type simply allows AMPS to work with data of a particular format.

In some cases, a given message type cannot support all of the capabilities in AMPS. For example, the unparsed `binary` message type allows arbitrary payloads. This can be extremely useful, but because there is no set format for that message type, none of the capabilities that rely on parsing data are supported by the `binary` message type. Where a message type cannot provide a specific capability to AMPS, those limitations are described below.

Message types in AMPS are implemented as plug-in modules. For more information on plug-in modules, contact 60East support for access to the AMPS Server SDK.

17.1. Default Message Types

AMPS automatically loads modules for the following message types:

Table 17.1. AMPS Default Message Types

Message Type Name	Description
`bson`	Binary JSON (BSON) messages. See http://www.bsonspec.org for information on this format.
`bflat`	BFlat, a schemaless message format based on key-value pairs that includes support for binary representations of numeric data.
`fix`	FIX messages using numeric tags. FIX is a standard format widely used in the financial industry. See http://www.fixtradingcommunity.org/pg/main/what-is-fix for more information on this format.
`json`	JSON (JavaScript Object Notation) messages. See http://www.json.org for information on this format.
`nvfix`	NVFIX messages. NVFIX uses the basic format as FIX, but allows arbitrary alphanumeric tags.
`xml`	XML messages (of any schema)
`binary`	Uninterpreted binary payload. Because this module does not attempt to parse the payload, it does not support content filtering, views and aggregates. Likewise, because there is no set format for the payload, this message type cannot support features that construct messages (such

Message Type Name	Description
	as delta messaging, `/AMPS/.*` topic subscriptions and `stats` acks).
`protobuf`	Google protocol buffer messages. To use this message type, you must configure a `MessageType` with the format of the messages (the `.proto` files).

With these message types, AMPS automatically loads the module that provides the message type. AMPS declares message types for all of the above message types except for `protobuf`.

For efficiency, AMPS only parses the content of a message if required, and only to the extent required. For example, if AMPS only needs to find the `id` tag in an NFVIX message, AMPS will not fully parse the message, but will stop parsing the message after finding the `id` tag. This provides significant performance improvements, and also means that AMPS does not verify the format or validity of messages unless it needs to parse the messages. When AMPS parses a message, it may only partially parse a message, and may not detect corruption or invalid format in a message if that corruption occurs after the point at which AMPS has all of the required information from the message.

The FIX and NVFIX message types support configuration of the field and message delimiters.

AMPS also allows you to create new message types by assembling existing message types into a *composite message*. Composite message types are described in Section 17.3, and require additional configuration:

Table 17.2. AMPS composite message types

Message Type Name	Description
`composite-global`	Composite message type that combines message parts for content filtering. This message type combines one or more existing message types into a message. This type is described in more detail in Section 17.3.
`composite-local`	Composite message type, filterable by individual parts. This message type combines one or more existing message types into a message. This type is described in more detail in Section 17.3.

17.2. BFlat Messages

The BFlat message format combines the simplicity and efficiency of simple, schema-less data formats such as FIX and NVFIX with the ability to manage binary data and preserve the full precision of numeric values. BFlat is especially useful for applications that deal with binary data or precise numeric values while demanding high levels of throughput.

A BFlat message is composed of any number of tag/value pairs, similar to FIX and NVFIX messages. Tags and values can contain any value, and can be of any length: unlike formats such as FIX, there are no reserved characters. In practical terms, the name of a tag must be a valid XPath identifier to filter the message in AMPS. However, this is a limitation of XPath, and not of the BFlat message format.

The BFlat message type supports all AMPS features, and there are no special considerations when using the BFlat message type.

BFlat Data Types

BFlat messages are strongly typed. BFlat supports a `string` type for string data, and a `binary` type for arbitrary binary data. For numeric values, BFlat can preserve the precise value of the following numeric types:

Table 17.3. BFlat Numeric Types

Type	Description
int8	8-bit integer
int16	16-bit integer
int32	32-bit integer
int64	64-bit integer
double	64-bit IEEE 754 floating point number
datetime	UTC datetime containing milliseconds since Unix epoch (64-bit representation)
leb128	Signed LEB128 integer (variable length)

BFlat also supports arrays of values.

17.3. Composite Messages

Sometimes, applications only need to filter on a small subset of the fields in a message. Sometimes applications need to send and receive messages that cannot be meaningfully parsed by AMPS, such as images or audio files. For these cases, AMPS provides a composite message type that lets you create a new message type by combining existing message types.

For example, you might create a message type that includes three parts: the metadata for an image as a `json` document, a small JPG thumbnail as a `binary` message part, and a full size PNG image as another `binary` message part.

Composite messages can also be useful when the message itself is large or resource-intensive to parse. In this case, you can create a message type that includes the information needed to filter messages in a JSON or NVFIX part, and include the full message in the unparsed payload of the composite message, as described below.

AMPS provides two different types of composite messages. Messages created using the `composite-local` module preserve information about the individual parts for filtering, aggregation, and projection. Messages creating using the `composite-global` module treat the individual parts as elements of a single document.

Configuring Composite Message Types

To use a composite message type, you must first configure the type by declaring it in the `MessageTypes` section of the AMPS configuration file. The declaration contains the name of the new composite message type, specifies that the new type is composite, and lists the parts of the composite message type.

For example, the `MessageType` element below declares a new composite message type named `images`. The new type contains a `json` document at the beginning of the message, followed by two uninterpreted binary message parts. AMPS will combine the XPath identifiers for all message parts into a single set of identifiers. Notice that, because only one part of the message type is parsable, using `composite-global` simplifies the identifiers for the message.

```
<MessageTypes>
...
  <MessageType>
    <Name>images</Name>
    <Module>composite-global</Module>
    <MessageType>json</MessageType>
    <MessageType>binary</MessageType>
    <MessageType>binary</MessageType>
  </MessageType>
...
</MessageTypes>
```

The `MessageType` entries for the composite message can be any AMPS message type, including any previously defined composite message.

Once the new composite message type is created, you can use the new type in the configuration file.

Composite message types have the following restrictions:

- Delta subscribe and delta publish are not supported for message types that use `composite-global`.
- Views, joins, and aggregation cannot project message types that use `composite-global`. (However, composite message types that use `composite-global` *can* be an `UnderlyingTopic` or one of the topics in a `Join`.)
- Composite message types do not support features that automatically construct messages, such as subscriptions the `AMPS/.*` topics and stats acks, regardless of the module the type uses.

Unparsed Payload Section

All composite message types, regardless of how they are defined, provide an *unparsed payload* section. The unparsed payload section does not need to be declared in the `MessageType` declaration. As the name suggests, AMPS does not parse or interpret this section, so the unparsed payload can contain any content of any type. The AMPS clients provide access to set the unparsed payload on outgoing messages, and to retrieve the unparsed payload from incoming messages.

The unparsed payload is included to simplify the common technique where a message type contains a header that is used for filtering followed by an unparsed binary. If your composite message type contains a single binary part, consider using the unparsed payload section in your application rather than declaring a binary message part.

Content Filtering with Composite Message Types

Composite message types support filtering on the contents of the composite message. There are some simple conventions to remember when constructing expressions to filter on. For more details about content filtering, see Section 4.2.

These conventions are consistent anywhere that AMPS needs to find a value within the composite message type. That includes content filters for client subscriptions, identifying SOW keys, creating views and aggregates, creating conflated topics, and so on.

composite-global

When using the `composite-global` message type, AMPS combines all parts of the message into a unified set of XPath identifiers. AMPS creates the set of identifiers for each part of the message. If different parts of the message contain the same identifier, AMPS treats that identifier as though the identifier contained an array of values: AMPS creates an array that contains all of the values in the different parts of the message. Message types that do not support content filtering do not provide XPath identifiers.

For example, consider the message below for a `composite-global` message type that includes two `json` parts and a `binary` part:

```
{"id":1,"data":"sample","message":"part one message"}
{"message":"another part","customer":"Awesome Amalgamated, Ltd."}
0xDEEA0934DF23A37780934...
```

AMPS constructs the following set of XPath identifiers and values:

Table 17.4. Composite-global message identifiers

Identifier	Value
`/id`	`1`
`/data`	`"sample"`
`/message`	`["part one message", "another part"]`
`/customer`	`"Awesome Amalgamated, Ltd."`

In short, when using `composite-global`, AMPS combines the parsable parts of the message into a single global set of XPath values, and ignores any part of the message that cannot be parsed.

composite-local

When using the `composite-local` message type, AMPS creates a distinct set of XPath identifiers for each part of the message. AMPS adds an XPath step with the position of the message part at the beginning of the identifier. Message types that do not support content filtering do not provide XPath identifiers, and AMPS skips over them.

For example, consider the message below for a `composite-local` message type that includes two `json` parts and a `binary` part:

```
{"id":1,"data":"sample","message":"part one message"}
{"message":"another part","customer":"Awesome Amalgamated, Ltd."}
0xDEEA0934DF23A37780934...
```

AMPS constructs the following set of XPath identifiers and values:

Table 17.5. Composite-local message identifiers

Identifier	Value
`/0/id`	`1`
`/0/data`	`"sample"`
`/0/message`	`"part one message"`
`/1/message`	`"another part"`

Identifier	Value
/1/customer	"Awesome Amalgamated, Ltd."

In short, when using composite-local, AMPS creates XPath identifiers for each part of the message, using the position of the message part within the composite as the first part of the identifier. AMPS skips over any part of the message that cannot be parsed, and simply produces no values for that part of the message.

Choosing A Composite Type

To choose which composite type best fits your application, consider the following factors:

- If you need to use delta messaging with this message type, use composite-local.
- If there may be redundant field names in the parts of the message, and it is important to be able to filter based on which part contains the field, use composite-local.
- If you need to be able to create views of this type, use composite-local.

Otherwise, composite-global may be easier and more straightforward for client filtering, since clients do not need to know the detailed structure of the message type to be able to filter on the message.

17.4. Protobuf Message Types

Protocol buffers, or protobufs for short, is an efficient, automated mechanism for serializing structured data. AMPS supports Google protobuf messages (version 2) as a message format.

Because Google protocol buffers use a fixed format for messages, to use protobuf, you must configure AMPS with the definition of the messages AMPS will process. This involves defining a MessageType. You must define a MessageType for AMPS to be able to parse protobuf messages.

Configuring Protobuf Message Types

To use a protobuf message, you must first edit the configuration file to include a new MessageType. Then, specify the path to the protobuf file and the file itself inside the MessageType. Below is a sample configuration of a protobuf message type:

```
...

<MessageType>
    <Name>my-protobuf-messages</Name>
    <Module>protobuf</Module>
    <ProtoPath>proto-archive;/mnt/shared/protofiles</ProtoPath>
    <ProtoFile>proto-archive/person.proto</ProtoFile>
    <Type>MyNamespace.Message</Type>
</MessageType>

...
```

When creating a protobuf message type, you must provide the following parameters:

Table 17.6. protobuf Message Type Parameters

Parameter	Description
`Name`	The name of the new, customized message type. The rest of the configuration file will use this name to refer to the message type.
`Module`	The module that contains the message type. Use `protobuf` for protocol buffer messages.

`ProtoPath`	The path in which to search for `.proto` files. The content of this element has the following syntax:
	`alias ; full-path`
	The alias provides a short identifier to use when searching for .proto files. The full path is the path that is substituted for that identifier.
	For example, in the sample above, `proto-archive` is an alias for `/mnt/shared/protofiles`.
	A configuration may omit the alias, and simply provide the path. For example:
	`;/mnt/repository/protodefs`
	You may specify any number of `ProtoPath` declarations.

`ProtoFile`	The name of the `.proto` file to use for this message type. To use an alias, prefix the name of the file with the alias, as shown in the example above.
`Type`	The name of the type inside the `.proto` file.

Filtering with Protobuf Messages

To filter protobuf messages, there are a couple of conventions you must remember. AMPS XPath identifiers begin at the outermost message, so you can simply use member names for that message. If you have nested messages, you use the name of the nested message and the member name when creating an XPath identifier.

For example, suppose you have the following `.proto` file:

```
message person {
  required string name = 1;
  required int32 personID = 2;
}
```

To access the `personID` data member, you simply use the name of the data member as the XPath identifier. An example filter that verifies that a `personID` is greater than 1000 would be:

```
/personID > 1000
```

If you have nested messages, you simply provide the path to the nested message you want to access.

Let's assume that that the `person` message from the above example was nested inside another message with the name of `record`. The example filter below shows how to access the nested `person` message, and then filter to the `personID`:

```
/person/personID > 1000
```

In this case, the first part of the identifier (`/person`) specifies the submessage. The second part of the identifier (`/personID`) specifies the field within that submessage. Notice that, as always, there is no need to specify the name of the message for the outermost message.

Union Types

When using a protocol buffer message type that contains a union, you can navigate the union using the names defined in the top-level element. For example, given the union defined below:

```
message MyUnion {
   optional Order      order_type = 1;
   optional Payment    payment_type = 2;
}

message Order {
 required string customer_id = 1;
 ...
}

message Payment {
  required string customer_id = 1;
  ...
}
```

Providing a filter of `/order_type IS NOT NULL` will return all of the `MyUnion` messages that contain an `Order`, while providing a filter of `/payment_type/customer_id = '42'` will return only the `MyUnion` messages that contain a `Payment` message with a `customer_id` of `42`.

Limitations of the protobuf message type

Because the `protobuf` message type requires a specific, fixed definition for messages, AMPS does not support operations that construct messages that may contain arbitrary values. In particular, protobuf does not support:

- Creating a View with `protobuf` as the `MessageType`. AMPS allows you to aggregate protobuf messages and project the results as another type, but the `MessageType` for a View cannot be a `protobuf` message type.
- Subscriptions to AMPS internal topics. Protobuf message types do not support creating messages for AMPS internal topics, such as `/AMPS/ClientStatus`.

Working with Optional Default Values

Google protocol buffers provide the ability for a message to have fields that are both *optional*, so they need not be provided in the serialized message, and *defaulted*, so that there is a specific value interpreted when there is no value provided.

When no value is provided in the serialized message for an optional default value, AMPS interprets the message differently depending on the context:

- For most uses, AMPS interprets the message as though the value is *present and set to the default value*. This means that you can filter on optional default values, use them as SOW keys, and aggregate optional default values regardless of whether a value is present in the serialized message.
- For *delta messaging*, AMPS treats an optional default value as though there is *no value present*. AMPS does not provide the default value. This means that a delta update must provide the default value *explicitly* in the serialized message to set the field to the default value. This also means that, if the value present in the message is not the default value, but was not changed on the current update, AMPS will not emit that value in messages to delta subscribers.

17.5. Loading Additional Message Types

AMPS includes the ability to load custom message types in external modules. As with all AMPS modules, custom message types are compiled into shared object files. AMPS dynamically loads these message types on startup, using the information provided in the configuration file. Once you have loaded and declared those types, you can use the type just as you use the default message types.

For example, the configuration below creates a message type named `custom-type` that uses a module named `libmy-type-module.so` and specifies a transport for messages of that type:

```
<Modules>
  <Module>
    ❶<Name>custom-type-module</Name>
    ❷<Library>./custom-modules/libmy-type-module.so</Library>
  </Module>
</Modules>

<MessageTypes>
  <MessageType>
    ❸<Name>custom-type</Name>
    ❹<Module>custom-type-module</Module>
  </MessageType>
</MessageTypes>

<Transports>
  <Transport>
    <Name>custom-type-tcp</Name>
    <Type>tcp</Type>
    <InetAddr>9008</InetAddr>
    ❺<MessageType>custom-type</MessageType>
    <Protocol>amps</Protocol>
  </Transport>
</Transports>
```

❶ Specifies the name to use to refer to this module in the rest of the configuation file

❷ Path to the library to load for this module. In this example, the path is a relative path below the directory where AMPS is started.

❸ The name to use for this message type in the rest of the configuration file.

❹ Reference to the module that implements this message type, using the `Name` defined in the `Module` configuration.

❺ The message type that this transport uses, using the `Name` defined in the `MessageType` configuration.

Once a message type has been declared, you can use it in exactly the same way you use the default message types.

Notice, however, that custom-developed message types may only provide support for a subset of the features of AMPS. For example, the `binary` message type provided with AMPS does not support features that require AMPS to parse or construct a message, as described above. The developer of the message type must provide information on what capabilities the message type provides.

Chapter 18. Transports

In order to send and receive messages, an AMPS server must allow incoming connections. *Transports* configure incoming connections to AMPS. Transports are configured in the `Transports` element of the AMPS configuration file.

AMPS provides two distinct kinds of incoming connections:

- *Client connections*, for use by the AMPS clients to support external applications
- *Replication connections*, to replicate to other AMPS instances

Each transport controls how authentication and entitlements are enforced for that transport. The transport can either accept the defaults for the instance as a whole, or choose settings unique to that transport.

18.1. Client connections

To accept connections from publishers or subscribers, an AMPS instance must have at least one Transport configured for client connections. The transport must specify:

- The network protocol used for the transport, called the transport *type*
- The AMPS command header format, called the *protocol*
- The network address, such as IP address and port, that the AMPS server will listen to for incoming connections

A transport can *optionally* set other parameters on the transport. This includes setting the authentication and entitlements that apply to connections for this transport, setting slow client parameters for the transport, and so forth.

SSL Connections

AMPS supports SSL connections between clients and servers. *In this release, SSL support is provided as a preview.* To enable SSL on a transport, you must:

- Specify a Transport type of `tcp` or `tcps`, *and*
- Provide a certificate and private key for the connection

You can optionally set other parameters for SSL connections, as described in the *AMPS Configuration Reference*.

> 60East recommends using the `tcps` transport type for SSL connections for clarity. However, AMPS uses SSL connections for a `tcp` connection whenever a PrivateKey and Certificate are provided for a Transport, regardless of whether the transport `Type` is specified as `tcp` or `tcps`.
>
> AMPS clients require that the connection string use `tcps` for SSL connections, even if the AMPS Transport configuration uses `tcp`.

18.2. Replication Connections

To receive replicated messages from other AMPS instances, an AMPS instance must have a transport configured as Type `amps-replication`.

Replication connections accept any message type, and can service multiple upstream AMPS instances.

Replication connections are configured as part of an overall High Availability plan. See Overview of AMPS High Availability and the AMPS Configuration Reference for details.

Part III. Deployment, Monitoring, and Administration

Chapter 19. Running AMPS as a Linux Service

AMPS is designed to be able to easily integrate into your existing infrastructure: AMPS includes all of the dependencies it needs to run, and is configured easily with a single configuration file. Some deployments integrate AMPS into a third-party service management infrastructure: for those deployments, the needs of that infrastructure determine how to install AMPS.

More typically, AMPS runs as a Linux service. This chapter describes how to install AMPS as a service.

19.1. Installing the Service

AMPS includes a shell script that installs the service. The shell script is included in the `bin` directory of your AMPS installation. Run the script with root permission, as follows:

```
$ sudo ./install-amps-daemon.sh
```

This script does the following installation work:

- Installs the AMPS distribution into `/opt/amps`
- Creates the `/opt/etc/amps` directory if it does not already exist. By default, the daemon uses an AMPS configuration file at `/opt/etc/amps/config.xml`.
- Installs the service management scripts. Depending on the init system the script detects on your system, this will either be a System V style script located at `/etc/init.d/amps` or a SystemD service definition file named `amps.service` installed under `/usr/lib/systemd/`.
- Updates the service management infrastructure to register AMPS as a service and configure the service to start on startup. The exact steps that the script takes to do this depend on the init system detected.

In addition, you must copy the AMPS configuration file for the instance to `/opt/etc/amps/config.xml`.

You can only run one instance of AMPS as a service on a system at a given time using this script. AMPS does not enforce any restriction on how many instances can be run on the system at the same time through other means, but this script is designed to manage a single instance running as a service.

19.2. Configuring the Service

When running as a service, the following considerations apply to the configuration file:

AMPS Logging

60East recommends logging the most important AMPS messages to syslog when running as a service. For example, the following configuration file snippet logs messages of warning level and above to the system log:

```
  <Logging>
    <Target>
```

```
      <Protocol>syslog</Protocol>
      <Level>warning</Level>
      <Ident>amps</Ident>
      <Options>LOG_CONS,LOG_NDELAY,LOG_PID</Options>
      <Facility>LOG_USER</Facility>
    </Target>
  </Logging>
```

60East does not recommend logging a level lower than warning to syslog, since an active AMPS instance can produce a large volume of messages.

File Paths

When running as a service, file paths in the configuration file also require attention. In particular:

- For simplicity, use absolute paths for all file paths in the configuration file.
- Consider startup order, and ensure that any devices that AMPS uses are mounted before AMPS starts.

  As with any other AMPS installation, it's also important to estimate the amount of storage space AMPS requires, and ensure that the device where AMPS stores files has the needed capacity.

Configuration File Location

The AMPS service scripts require the configuration file to be located at `/opt/etc/amps/config.xml`.

19.3. Managing the Service

The scripts that AMPS installs provide management functions for the AMPS service. The scripts are used in the same way scripts for other Linux services are used.

Starting the AMPS Service

To start the AMPS service, use the following command if your system uses System V-style init scripts:

```
sudo /etc/init.d/amps start
```

Many systems that use System V init scripts also provide convenience commands (such as `service`) to locate and run commands for working with daemons. Check your distribution's documentation for details.

If your system uses SystemD, you can use a command like:

```
sudo systemctl start amps
```

Stopping the AMPS Service

To stop the AMPS service, use the following command if your system uses System V init scripts:

```
sudo /etc/init.d/amps stop
```

Many distributions that use System V init scripts also provide convenience commands (such as the `service` program) for working with daemons. Check your distribution's documentation for details.

If your system uses SystemD, you can use a command like:

```
sudo systemctl stop amps
```

Restarting the AMPS Service

To restart the AMPS service, use the following command if your system uses System V init scripts:

```
sudo /etc/init.d/amps restart
```

Many distributions that use System V init scripts also provide convenience commands (such as the `service` program) for working with daemons. Check your distribution's documentation for details.

If your distribution uses SystemD, you can use a command like:

```
sudo systemctl restart amps
```

View status for the AMPS Service

To see the status of the AMPS service, use the following command if your distribution uses System V init scripts:

```
sudo /etc/init.d/amps status
```

Many distributions that use System V init scripts also provide convenience commands (such as the `service` program) for working with daemons. Check your distribution's documentation for details.

If your distribution uses SystemD, you can use a command like:

```
sudo systemctl status amps
```

19.4. Uninstalling the Service

AMPS includes a script that uninstalls AMPS as a service. The script reverses the changes that the install script makes to your system. Run the script with root permission, as follows:

```
$ sudo ./uninstall-amps-daemon.sh
```

The uninstall script does not remove the configuration file or any files or data that AMPS creates at runtime.

19.5. Upgrading the Service

To upgrade the service to a new version of AMPS, follow these steps:

1. Stop the service.

2. Uninstall the previous version of the service using the uninstall script included with that version.

3. If necessary, upgrade any data files or configuration files that you want to retain.

4. Install the new version of the service using the install script included with the new version. Ensure that the configuration file is at the appropriate path for the new installation.

5. Start the service.

For AMPS instances that participate in failover, you must coordinate your upgrades as you would for a standalone AMPS instance.

Chapter 20. Logging

AMPS supports logging to many different targets including the console, syslog, and files. Every error message within AMPS is uniquely identified and can be filtered out or explicitly included in the logger output. This chapter of the *AMPS User Guide* describes the AMPS logger configuration and the unique settings for each logging target.

20.1. Configuration

Logging within AMPS is enabled by adding a Logging section to the configuration. For example, the following would log all messages with an 'info' level or higher to the console:

```
<AMPSConfig>
  ...
  <Logging>
   ❶<Target>
      <Protocol>stdout</Protocol>
     ❷<Level>info</Level>
    </Target>
  </Logging>
  ...
</AMPSConfig>
```

❶ The `Logging` section defines a single `Target`, which is used to log all messages to the `stdout` output.
❷ States that only messages with a log level of `info` or greater will be output to the screen.

20.2. Log Messages

An AMPS log message is composed of the following:

- Timestamp (eg: 2010-04-28T21:52:03.4766640-07:00)
- AMPS thread identifier
- Log Level (eg: info)
- Error identifier (eg: 15-0008)
- Log message

An example of a log line (it will appear on a single line within the log):

```
2011-11-23T14:49:38.3442510-08:00 [1] info: 00-0015 AMPS initialization
 completed (0 seconds).
```

Each log message has a unique identifier of the form `TT-NNNN` where `TT` is the component within AMPS which is reporting the message and `NNNN` the number that uniquely identifies that message within the module. Each logging target allows the direct exclusion and/or inclusion of error messages by identifier. For example, a log file which would include all messages from module `00` except for `00-0001` and `00-0004` would use the following configuration:

```
<Logging>
  <Target>
    <Protocol>stdout</Protocol>
    <IncludeErrors>00-0002</IncludeErrors>
    <ExcludeErrors>00-0001,00-0004,12-1.*</ExcludeErrors>
  </Target>
</Logging>
```

The above `Logging` configuration example, all log messages which are at or above the default log level of `info` will be emitted to the logging target of `stdout`. The configuration explicitly wants to see configuration messages where the error identifier matches `00-0002`. Additionally, the messages which match `00-0001`, `00-0004` will be excluded, along with any message which match the regular expression of `12-1.*`.

20.3. Log Levels

AMPS has nine log levels of escalating severity. When configuring a logging target to capture messages for a specific log level, all log levels at or above that level are sent to the logging target. For example, if a logging target is configured to capture at the “error” level, then all messages at the “error”, “critical”, and “emergency” levels will be captured because “critical” and “emergency” are of a higher level. The following table Table 20.1 contains a list of all the log levels within AMPS.

Table 20.1. Log Levels

Level	Description
none	no logging
developer	information on the internal state of AMPS
trace	all inbound/outbound data
debug	**Obsolete**. The AMPS server no longer logs messages at this level. Plugin modules that attempt to log messages at this level will log messages at info level instead.
stats	statistics messages
info	general information messages
warning	problems that AMPS tries to correct that are often harmless
error	events in which processing had to be aborted
critical	events impacting major components of AMPS that if left uncorrected may cause a fatal event or message loss
emergency	a fatal event

Each logging target allows the specification of a `Level` attribute that will log all messages at the specified log level or with higher severity. The default `Level` is `none` which would log nothing. Optionally, each target also allows the selection of specific log levels with the `Levels` attribute. Within `Levels`, a comma separated list of levels will be additionally included.

For example, having a log of only `trace` messages may be useful for later playback, but since `trace` is at the lowest level in the severity hierarchy it would normally include all log messages. To only enable `trace` level, specify `trace` in the `Levels` setting as below:

```
<AMPSConfig>
  ...
  <Logging>
    <Target>
      <Protocol>gzip</Protocol>
      <FileName>traces.log.gz</FileName>
      <Levels>trace</Levels>
    </Target>
  </Logging>
  ...
</AMPSConfig>
```

Logging only `trace` and `info` messages to a file is demonstrated below:

```
<AMPSConfig>
  ...
  <Logging>
    <Target>
      <Protocol>file</Protocol>
      <FileName>traces-info.log</FileName>
      <Levels>trace,info</Levels>
    </Target>
  </Logging>
  ...
</AMPSConfig>
```

Logging `trace`, `info` messages in addition to levels of `error` and above (`error`, `critical` and `emergency`) is demonstrated below:

```
<Target>
   <Protocol>file</Protocol>
   <FileName>traces-error-info.log</FileName>
   <Level>error</Level>
   <Levels>trace,info</Levels>
</Target>
```

20.4. Logging to a File

To log to a file, declare a logging target with a protocol value of `file`. Beyond the standard `Level`, `Levels`, `IncludeErrors`, and `ExcludeErrors` settings available on every logging target, file targets also permit the selection of a `FileName` mask and `RotationThreshold`.

Selecting a Filename

The `FileName` attribute is a mask which is used to construct a directory and file name location for the log file. AMPS will resolve the file name mask using the symbols in Table 20.2. For example, if a file name is masked as:

```
%Y-%m-%dT%H:%M:%S.log
```

...then AMPS would create a log file in the current working directory with a timestamp of the form: `2012-02-23T12:59:59.log`.

If a `RotationThreshold` is specified in the configuration of the same log file, the the next log file created will be named based on the current system time, not on the time that the previous log file was generated. Using the previous log file as an example, if the first rotation was to occur 10 minutes after the creation of the log file, then that file would be named `2012-02-23T13:09:59.log`.

Log files which need a monotonically increasing counter when log rotation is enabled can use the `%n` mask to provide this functionality. If a file is masked as:

```
localhost-amps-%n.log
```

Then the first instance of that file would be created in the current working directory with a name of `localhost-amps-00000.log`. After the first log rotation, a log file would be created in the same directory named `localhost-amps-00001.log`.

Log file rotation is discussed in greater detail in the section called “Log File Rotation”.

Table 20.2. Log Filename Masks

Mask	Definition
`%Y`	Year
`%m`	Month
`%d`	Day
`%H`	Hour
`%M`	Minute
`%S`	Second
`%n`	Iterator which starts at `00000` when AMPS is first started and increments each time a `RotationThreshold` size is reached on the log file.

Log File Rotation

Log files can be “rotated” by specifying a valid threshold in the `RotationThreshold` attribute. Values for this attribute have units of bytes unless another unit is specified as a suffix to the number. The valid unit suffixes are:

Table 20.3. Log File Rotation Units

Unit Suffix	Base Unit	Examples
no suffix	bytes	“1000000” is 1 million bytes
k or K	thousands of bytes	“50k” is 50 thousand bytes

Unit Suffix	Base Unit	Examples
m or M	millions of bytes	“10M” is 10 million bytes
g or G	billions of bytes	“2G” is 2 billion bytes
t or T	trillions of bytes	“0.5T” is 500 billion bytes

When using log rotation, if the next filename is the same as an existing file, the file will be truncated before logging continues. For example, if “amps.log” is used as the `FileName` mask and a `RotationThreshold` is specified, then the second rotation of the file will overwrite the first rotation. If it is desirable to keep all logging history, then it is recommended that either a timestamp or the `%n` rotation count be used within the `FileName` mask when enabling log rotation.

Examples

The following logging target definition would place a log file with a name constructed from the timestamp and current log rotation number in the `./logs` subdirectory. The first log would have a name similar to `./logs/20121223125959-00000.log` and would store up to 2GB before creating the next log file named `./logs/201212240232-00001.log`.

```
<AMPSConfig>
  ...
  <Logging>
    <Target>
      <Protocol>file</Protocol>
      <Level>info</Level>
      <FileName>./logs/%Y%m%d%H%M%S-%n.log</FileName>
      <RotationThreshold>2G</RotationThreshold>
    </Target>
  </Logging>
  ...
</AMPSConfig>
```

This next example will create a single log named `amps.log` which will be appended to during each logging event. If `amps.log` contains data when AMPS starts, that data will be preserved and new log messages will be appended to the file.

```
<AMPSConfig>
  ...
  <Logging>
    <Target>
      <Protocol>file</Protocol>
      <Level>info</Level>
      <FileName>amps.log</FileName>
    </Target>
  </Logging>
  ...
</AMPSConfig>
```

20.5. Logging to a Compressed File

AMPS supports logging to compressed files as well. This is useful when trying to maintain a smaller logging footprint. Compressed file logging targets are the same as regular file targets except for the following:

- the `Protocol` value is `gzip` instead of `file`;
- the log file is written with gzip compression;
- the `RotationThreshold` is metered off of the uncompressed log messages;
- makes a trade off between a small increase in CPU utilization for a potentially large savings in logging footprint.

Example

The following logging target definition would place a log file with a name constructed from the timestamp and current log rotation number in the `./logs` subdirectory. The first log would have a name similar to `./logs/20121223125959-0.log.gz` and would store up to 2GB of uncompressed log messages before creating the next log file named `./logs/201212240232-1.log.gz`.

```
<AMPSConfig>
  ...
  <Logging>
    <Target>
      <Protocol>gzip</Protocol>
      <Level>info</Level>
      <FileName>./logs/%Y%m%d%H%M%S-%n.log.gz</FileName>
      <RotationThreshold>2G</RotationThreshold>
    </Target>
  </Logging>
  ...
</AMPSConfig>
```

20.6. Logging to the Console

The console logging target instructs AMPS to log certain messages to the console. Both the standard output and standard error streams are supported. To select standard out use a `Protocol` setting of `stdout`. Likewise, for standard error use a `Protocol` of `stderr`.

Example

Below is an example of a console logger that logs all messages at the `info` or `warning` level to standard out and all messages at the `error` level or higher to standard error (which includes `error`, `critical` and `emergency` levels).

```
<AMPSConfig>
  ...
  <Logging>
    <Target>
      <Protocol>stdout</Protocol>
      <Levels>info,warning</Levels>
    </Target>
    <Target>
      <Protocol>stderr</Protocol>
      <Level>error</Level>
    </Target>
  </Logging>
  ...
</AMPSConfig>
```

20.7. Logging to Syslog

AMPS can also log messages to the host's syslog mechanism. To use the syslog logging target, use a `Protocol` of `syslog` in the logging target definition.

The host's syslog mechanism allows a logger to specify an identifier on the message. This identifier is set through the `Ident` property and defaults to the AMPS instance name (see *AMPS Configuration Reference Guide* for configuration of the AMPS instance name.)

The syslog logging target can be further configured by setting the `Options` parameter to a comma-delimited list of syslog flags. The recognized syslog flags are:

Table 20.4. Logging Options Available for SYSLOG Configuration

Level	Description
`LOG_CONS`	Write directly to system console if there is an error while sending to system logger.
`LOG_NDELAY`	Open the connection immediately (normally, the connection is opened when the first message is logged).
`LOG_NOWAIT`	No effect on Linux platforms.
`LOG_ODELAY`	The converse of LOG_NDELAY; opening of the connection is delayed until `syslog()` is called. (This is the default, and need not be specified.)
`LOG_PERROR`	Print to standard error as well.
`LOG_PID`	Include PID with each message.

AMPS already includes the process identifier (PID) with every message it logs, however, it is a good practice to set the `LOG_PID` flag so that downstream syslog analysis tools will find the PID where they expect it.

The `Facility` parameter can be used to set the syslog "facility". Valid options are: `LOG_USER` (the default), `LOG_LOCAL0`, `LOG_LOCAL1`, `LOG_LOCAL2`, `LOG_LOCAL3`, `LOG_LOCAL4`, `LOG_LOCAL5`, `LOG_LOCAL6`, or `LOG_LOCAL7`.

Finally, AMPS and the syslog do not have a perfect mapping between their respective log severity levels. AMPS uses the following table to convert the AMPS log level into one appropriate for the syslog:

Table 20.5. Comparison of AMPS Log Severity to Syslog Severity

AMPS Severity	Syslog Severity
none	LOG_DEBUG
developer	LOG_DEBUG
trace	LOG_DEBUG
debug	LOG_DEBUG
stats	LOG_INFO
info	LOG_INFO
warning	LOG_WARNING
error	LOG_ERR
critical	LOG_CRIT
emergency	LOG_EMERG

Example

Below is an example of a syslog logging target that logs all messages at the `critical` severity level or higher and additionally the log messages matching `30-0001` to the syslog.

```
<AMPSConfig>
  ...
  <Logging>
    <Target>
      <Protocol>syslog</Protocol>
      <Level>critical</Level>
      <IncludeErrors>30-0000</IncludeErrors>
      <Ident>\amps dma</Ident>
      <Options>LOG_CONS,LOG_NDELAY,LOG_PID</Options>
      <Facility>LOG_USER</Facility>
    </Target>
  </Logging>
  ...
</AMPSConfig
```

20.8. Error Categories

In the AMPS log messages, the error identifier consists of an error category, followed by a hyphen, followed by an error identifier. The error categories cover the different modules and features of AMPS, and can be helpful in diagnostics and troubleshooting by providing some context about where a message is being logged from. A list of the error categories found in AMPS are listed in Table 20.6.

Table 20.6. AMPS Error Categories

AMPS Code	Component
00	AMPS Startup
01	General
02	Message Processing
03	Expiration
04	Publish Engine
05	Statistics
06	Metadata
07	Client
08	Regex
09	ID Generator
0A	Diff Merge
0B	Out of Focus processing
0C	View
0D	Message Data Cache
0E	Conflated Topic
0F	Message Processor Manager
11	Connectivity
12	Trace In - for inbound messages
13	Datasource
14	Subscription Manager
15	SOW
16	Query
17	Trace Out - for outbound messages
18	Parser
19	Administration Console
1A	Evaluation Engine
1B	SQLite
1C	Meta Data Manager
1D	Transaction Log Monitor
1E	Replication
1F	Client Session
20	Global Heartbeat
21	Transaction Replay
22	TX Completion
23	Bookmark Subscription
24	Thread Monitor
25	Authorization

AMPS Code	Component
26	SOW cache
28	Memory cache
29	Authorization & entitlement plugins
2A	Message pipeline
2B	Module manager
2C	File management
2D	NUMA module
2F	SOW update broadcaster
30	AMPS internal utilities
70	AMPS networking
FF	Shutdown

20.9. Looking Up Errors with ampserr

In the `$AMPSDIR/bin` directory is the `ampserr` utility. Running this utility is useful for getting detailed information and messages about specific AMPS errors observed in the log files.

The *AMPS Utilities User Guide* contains more information on using the `ampserr` utility and other debugging tools.

Chapter 21. Event Topics

AMPS publishes specific events to internal topics that begin with the /AMPS/ prefix, which is reserved for AMPS only. For example, all client connectivity events are published to the internal /AMPS/ClientStatus topic. This allows all clients to monitor for events that may be of interest.

Event topic messages can be subscribed with content filters like any other topic within AMPS.

A client may subscribe to event topics on any connection with a message type that supports views. This includes all of the default message types and bson, but does not include the binary message type.

Messages are delivered as the message type for the connection. For example, if the connection uses JSON messages, the event topic messages with be JSON. A connection that uses FIX will receive FIX messages from an event topic.

21.1. Client Status

The AMPS engine will publish client status events to the internal /AMPS/ClientStatus topic whenever a client issues a logon command, disconnects, enters or removes a subscription, queries a SOW, or issues a sow_delete. AMPS sends a message if a client fails authentication. In addition, upon a disconnect, a client status message will be published for each subscription that the client had registered at the time of the disconnect. This mechanism allows any client to monitor what other clients are doing and is especially useful for publishers to determine when clients subscribe to a topic of interest.

To help identify clients, it is recommended that clients send a logon command to the AMPS engine and specify a meaningful client name. This client name is used to identify the client within client status event messages, logging output, and information on clients within the monitoring console. The client name must be unique if a transaction log is configured for the AMPS instance.

Each message published to the client status topic will contain an Event and a ClientName. For subscribe and unsubscribe events, the message will contain Topic, Filter and SubId.

When the connection uses the xml message type, the client status message published to the /AMPS/ClientStatus will contain a SOAP body with a ClientStatus section as follows:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header>
    <Cmd>publish</Cmd>
      <TxmTm>20090106-23:24:40-0500</TxmTm>
      <Tpc>/AMPS/ClientStatus</Tpc>
      <MsgId>MAMPS-55</MsgId>
      <SubId>SAMPS-1233578540_1</SubId>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
      <ClientStatus>
        <Event>subscribe</Event>
        <ClientName>test_client</ClientName>
        <Topic>order</Topic>
```

```
        <Filter>(/FIXML/Order/Instrmt/@Sym = 'IBM')</Filter>
        <SubId>SAMPS-1233578540_10</SubId>
      </ClientStatus>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Table 21.1 defines the header fields which may be returned as part of the subscription messages to the /AMPS/ClientStatus topic.

Table 21.1. /AMPS/ClientStatus Event Message Fields

FIX	XML	JSON / BSON	Definition
20065	Timestamp	timestamp	Timestamp at which AMPS processed the message
20066	Event	event	Command executed by the client
20067	ClientName	client_name	Client Name
20068	Tpc	topic	Topic for the event (if applicable)
20069	Filter	filter	Filter (if applicable)
20070	SubId	sub_id	Subscription ID (if applicable)
20071	ConnName	connection_name	Internal AMPS connection name
20072	Options	options	The options for the subscription (if applicable)
20073	QId	query_id	The identifier for the query (if applicable)
20080	ClientAddr	client_address	The remote address of the client
20081	AuthId	auth_id	The authenticated identity of the client (if applicable)

21.2. SOW Statistics

AMPS can publish SOW statistics for each SOW topic which has been configured. To enable this functionality, specify the SOWStatsInterval in the configuration file. The value provided in SOWStatsInterval is the time between updates to the /AMPS/SOWStats topic.

For example, the following would be a configuration that would publish /AMPS/SOWStats event messages every 5 seconds.

```
<AMPSConfig>
  ...
  <SOWStatsInterval>5s</SOWStatsInterval>
  ...
</AMPSConfig>
```

When receiving from the AMPS engine using the xml protocol, the SOW status message published to the /AMPS/SOWStats topic will contain a SOAP body with a SOWStats section as follows:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/">
  <SOAP-ENV:Header>
    <Cmd>publish</Cmd>
    <TxmTm>2010-09-08T17:49:06.9439120Z</TxmTm>
    <Tpc>/AMPS/SOWStats</Tpc>
    <SowKey>18446744073709551615</SowKey>
    <MsgId>AMPS-10548998</MsgId>
    <SubIds>SAMPS-1283968028_2</SubIds>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <SOWStats>
      <Timestamp>2010-09-08T17:49:06.9439120Z</Timestamp>
      <Topic>MyTopic</Topic>
      <Records>10485760</Records>
    </SOWStats>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In the `SOWStats` message, the `Timestamp` field includes the time the event was generated, `Topic` includes the topic, and `Records` includes the number of records.

Table 21.2 defines the header fields which may be returned as part of the subscription messages to the `/AMPS/SOWStats` topic.

Table 21.2. `/AMPS/SOWStats` Event Message Default Fields

FIX	XML	JSON/BSON	Definition
20065	`Timestamp`	`timestamp`	Timestamp in which AMPS sent the message
20066	`Topic`	`topic`	Topic that statistics are being reported on
20067	`Records`	`record_count`	Number of records in the SOW topic

For compatibility with systems that expect a consistent set of FIX tags across messages, AMPS provides a set of FIX tags that are unified with the tags used in the `/AMPS/ClientStatus` topic. To use the unified FIX tags, set the `AMPSVersionCompliance` configuration element to `5`. The following table lists the unified FIX tags:

Table 21.3. /AMPS/SOWStats Event Message Unified Fields (Version 5)

FIX	Definition
20064	Timestamp in which AMPS sent the message
20068	Topic that statistics are being reported on
20075	Number of records in the SOW topic

21.3. Persisting Event Topic Data

By default, AMPS event topics are not persisted to the SOW. However, because AMPS event topic messages are treated the same as all other messages, the event topics can be persisted to the SOW. Providing a topic definition with the appropriate `Key` definition can resolve that issue by instructing AMPS to persist the messages.

The `Key` definition you specify must match the field name used for the message type specified in the SOW topic. That is, to track distinct records by client name for a SOW that uses `json`, you would use the following key:

```
<Key>/client_name</Key>
```

While to track distinct records by client name for a SOW that uses `fix`, you would use the following key:

```
<Key>/20067</Key>
```

For example, to persist the last `/AMPS/SOWStats` message for each topic in `fix`, `xml` and `json` format, the following `Topic` sections could be added to the `SOW` section of the AMPS configuration file:

```
<SOW>

  <!-- Persist /AMPS/SOWStats in FIX format -->
  <Topic>
    <Name>/AMPS/SOWStats</Name>
    <FileName>./sow/sowstats.fix.sow</FileName>
    <MessageType>fix</MessageType>
    <!-- use FIX field for the key -->
    <Key>/20066</Key>
  </Topic>

  <!-- Persist /AMPS/SOWStats in JSON format -->
  <Topic>
     <Name>/AMPS/SOWStats</Name>
     <FileName>./sow/sowstats.json.sow</FileName>
     <MessageType>json</MessageType>
     <!-- use the JSON field for the key -->
     <Key>/topic</Key>
  </Topic>

  <!-- Persist /AMPS/SOWStats in XML format -->
  <Topic>
    <Name>/AMPS/SOWStats</Name>
    <FileName>./sow/sowstats.xml.sow</FileName>
    <MessageType>xml</MessageType>
     <!-- use the XML field for the key -->
    <Key>/Topic</Key>
  </Topic>
</SOW>
```

Every time an update occurs, AMPS will persist the `/AMPS/SOWStats` message and it will be stored three times, once to the `fix` SOW topic, once to the `xml` SOW topic, and once to the `json` SOW topic. Each update to the respective SOW topic will overwrite the record with the same `Topic`, `topic` or `20066` tag value. Doing this allows clients to now query the `SOWStats` topic instead of actively listening to live updates.

Chapter 22. Utilities

AMPS provides several utilities that are not essential to message processing, but can be helpful in troubleshooting or tuning an AMPS instance. Some of the most commonly-used utilities are listed below. Each of the following utilities is covered in greater detail in the *AMPS Utilities Guide*:

- `amps_upgrade` upgrades data files for existing AMPS instances to the current release of AMPS. 60East recommends running this utility whenever an upgrade changes either the major or minor version number (for example, an upgrade from 3.8.0.0 to 3.9.0.0).
- AMPS provides a command-line client, `spark`, as a useful tool for diagnostics, such as checking the contents of a SOW topic. The `spark` client can also be used for simple scripting to run queries, place subscriptions, and publish data.
- `ampserr` is used to expand and examine error messages that may be observed in the logs. This utility allows a user to input a specific error code, or a class of error codes, examine the error message in more detail, and where applicable, view known solutions to similar issues.
- `amps_sqlite3` provides a more convenient interface for querying AMPS statistics databases.
- `amps_sow_dump` is used to inspect the contents of a SOW topic store.
- `amps_journal_dump` is used to examine the contents of an AMPS journal file during debugging and program tuning.
- `amps_file` is used for identifying the filetype and version of files that AMPS persists (for example, `AMPS sow 6.0` for a SOW that uses version 6 of the SOW file format).

Chapter 23. Monitoring Interface

AMPS includes a monitoring interface which is useful for examining many important aspects about an AMPS instance. This includes health and monitoring information for the AMPS engine as well as the host AMPS is running on. All of this information is designed to be easily accessible to make gathering performance and availability information from AMPS easy. The monitoring interface also provides easy access to perform administrative actions.

The information in the monitoring database is taken from the statistics database for the AMPS instance. AMPS provides actions for managing the statistics database, as described in Section 24.11.

For a reference regarding the fields and their data types available in the AMPS monitoring interface, see the *AMPS Monitoring Reference*

23.1. Configuration

The AMPS monitoring interface is defined in the configuration file used on AMPS start up. Below is an example configuration of the `Admin` tag.

```
<!-- Configure the admin/stats HTTP server -->
<Admin>
   <FileName>stats.db</FileName>
   <InetAddr>localhost:8085</InetAddr>
   <Interval>10s</Interval>
</Admin>
```

In this example `localhost` is the hostname and `8085` is the port assigned to the monitoring interface. This chapter will assume that

`http://localhost:8085/`

is configured as the monitoring interface URL.

The `Interval` tag is used to set the update interval for the AMPS monitoring interface. In this example, statistics will be updated every 10 seconds.

> It is important to note that by default AMPS will store the monitoring interface database information in system memory. If the AMPS instance is going to be up for a long time, or the monitoring interface statistics interval will be updated frequently, it is strongly recommended that the `FileName` setting be specified to allow persistence of the data to a local file. See the *AMPS Configuration Reference Guide* for more information.

The administrative console is accessible through a web browser, but also follows a Representational State Transfer (RESTful) URI style for programmatic traversal of the directory structure of the monitoring interface.

The root of the AMPS monitoring interface URI contains two child resources—the `host` URI and the `instance` URI —each of which is discussed in greater detail below. The `host` URI exposes information about the current operating system devices, while the `instance` URI contains statistics about a specific AMPS deployment.

23.2. Time Range Selection

AMPS keeps a history of the monitoring interface statistics, and allows that data to be queried. By selecting a leaf node of the monitoring interface resources, a time-based query can be constructed to view a historical report of the information. For example, if an administrator wanted to see the number of messages per second consumed by all processors from midnight UTC on October 12, 2011 until 23:25:00 UTC on October 10, 2011, then pointing a browser to

`http://localhost:8085/amps/instance/processors/all/messages_received per_sec?t0=20111129T0&t1=20111129T232500`

will generate the report and output it in the following plain text format (note: entire dataset is not presented, but is truncated).

```
20111130T033400,0
20111130T033410,0
20111130T033420,0
20111130T033430,94244
20111130T033440.000992,304661
20111130T033450.000992,301078
20111130T033500,302755
20111130T033510,308922
20111130T033520.000992,306177
20111130T033530.000992,302140
20111130T033540.000992,302390
20111130T033550,307637
20111130T033600.000992,310109
20111130T033610,309888
20111130T033620,299993
20111130T033630,310002
20111130T033640.000992,300612
20111130T033650,299387
```

All times used for the report generation and presentation are ISO- 8601 formatted. ISO-8601 formatting is of the following form: `YYYYMMDDThhmmss`, where `YYYY` is the year, `MM` is the month, `DD` is the year, `T` is a separator between the date and time, `hh` is the hours, `mm` is the minutes and `ss` is the seconds. Decimals are permitted after the `ss` units.

As discussed in the following sections, the date-time range can be used with plain text (html), comma-separated values (csv), and XML formats.

23.3. Output Formatting

The AMPS monitoring interface offers several possible output formats to ease the consumption of monitoring reporting data. The possible options are XML, CSV and RNC output formats, each of which is discussed in more detail below.

XML Document Output

All monitoring interface resources can have the current node, along with all child nodes list its output as an XML document by appending the `.xml` file extension to the end of the resource name. For example, if an administrator would like to have an XML document of all of the currently running processors—including all the relevant statistics about those processors—then the following URI will generate that information:

```
http://localhost:8085/amps/instance/processors/all.xml
```

The document that is returned will be similar to the following:

```
<amps>
  <instance>
    <processors>
      <processor id='all'>
        <denied_reads>0</denied_reads>
        <denied_writes>0</denied_writes>
        <description>AMPS Aggregate Processor Stats</description>
        <last_active>1855</last_active>
        <matches_found>0</matches_found>
        <matches_found_per_sec>0</matches_found_per_sec>
        <messages_received>0</messages_received>
        <messages_received_per_sec>0</messages_received_per_sec>
        <throttle_count>0</throttle_count>
      </processor>
    </processors>
  </instance>
</amps>
```

Appending the `.xml` file extension to any AMPS monitoring interface resource will generate the corresponding XML document.

CSV Document Output

Similar to the XML document output discussed above, the `.csv` file extension can be appended to any of the leaf node resources to have a CSV file generated to examine those values. This can also be coupled with the time range selection to generate reports. See Section 23.2 for more details on time range selection.

Below is a sample of the `.csv` output from the monitoring interface from the following URL:

```
http://localhost:8085/amps/instance/processors/all/
matches_found_per_sec.csv?t0=20111129T0
```

This resource will create a file with the following contents:

```

20111130T033400,0
20111130T033410,0
20111130T033420,0
20111130T033430,94244
20111130T033440.000992,304661
20111130T033450.000992,301078
```

```
20111130T033500,302755
20111130T033510,308922
20111130T033520.000992,306177
20111130T033530.000992,302140
20111130T033540.000992,302390
20111130T033550,307637
20111130T033600.000992,310109
20111130T033610,309888
20111130T033620,299993
20111130T033630,310002
20111130T033640.000992,300612
20111130T033650,299387
20111130T033700.000992,304548
```

JSON Document Output

All monitoring interface resources can have the current node, along with all child nodes list its output as an JSON document by appending the `.json` file extension to the end of the resource name. For example, if an administrator would like to have an JSON document of all of the CPUs on the server—including all the relevant statistics about those CPUs—then the following URI will generate that information:

`http://localhost:8085/amps/host/cpus.json`

The document that is returned will be similar to the following:


```
{
  "amps":{
    "host":{
      "cpus":[
         {"id":"all"
          ,"idle_percent":"62.452316076294"
          ,"iowait_percent":"0.490463215259"
          ,"system_percent":"10.681198910082"
          ,"user_percent":"26.376021798365"
        }
        ,{"id":"cpu0"
          ,"idle_percent":"75.417130144605"
          ,"iowait_percent":"0.333704115684"
          ,"system_percent":"7.563959955506"
          ,"user_percent":"16.685205784205"
        }
        ,{"id":"cpu1"
          ,"idle_percent":"50.000000000000"
          ,"iowait_percent":"0.642398286938"
          ,"system_percent":"13.597430406852"
          ,"user_percent":"35.760171306210"
        }
      ]
    }
  }
}
```

Appending the `.json` file extension to any AMPS monitoring interface resource will generate the corresponding JSON document.

RNC Document Output

AMPS supports generation of an XML schema via the Relax NG Compact (RNC) specification language. To generate an RNC file, enter the following URL in a browser `http://localhost:port/amps.rnc` and AMPS will display the RNC schema.

To convert the RNC schema into an XML schema, first save the RNC output to a file:

```
%> wget http://localhost:9090/amps.rnc
```

The output can then be converted to an xml schema using Trang (available at http://code.google.com/p/jing-trang/) with

```
trang -I rnc -O xsd amps.rnc amps.xsd
```

Chapter 24. Automating AMPS With Actions

AMPS provides the ability to run scheduled tasks or respond to events, such as Linux signals, using the Actions interface.

To create an action, you add an `Actions` section to the AMPS configuration file. Each `Action` contains one (or more) `On` statement which specifies when the action occurs, and one (or more) `Do` statement which specifies what the AMPS server does for the action. Within an action, AMPS performs each `Do` statement in the order in which they appear in the file.

AMPS actions may require the use of parameters. AMPS allows you to use variables in the parameters of an action. You can access these variables using the following syntax:

`{{VARIABLE_NAME}}`

AMPS defines a set of default variables when running an action. The event, or a previous action, can add variables in the context of the action. Those variables can be expanded in subsequent parameters. If a variable is used that isn't defined at the point where it is used, AMPS will expand that variable to an empty string literal. The context can also be updated as the module is running, so any variables that are available at any given point in the file depend on what action was previously executed.

By default, AMPS loads the following variables when it initializes an AMPS action:

Table 24.1. Default Context Variables

Variable	Description
`AMPS_INSTANCE_NAME`	The name of the AMPS instance.
`AMPS_BYTE_XX`	Insert byte *XX*, where *XX* is a 2-digit uppercase hex number (00-FF). AMPS expands this variable to the corresponding byte value. These variables are useful for creating field separators or producing characters that are not permitted within XML
`AMPS_DATETIME`	The current date and time in ISO-8601 format.
`AMPS_UNIX_TIMESTAMP`	The current date and time as a UNIX timestamp.

An example to echo a message when AMPS starts up is shown below. Note the AMPS_INSTANCE_NAME is one of the variables that AMPS pushes to the context when an action is loaded.

```
<Actions>
  <Action>
    <On>
      <Module>amps-action-on-startup</Module>
    </On>
    <Do>
      <Module>amps-action-do-echo-message</Module>
      <Options>
        <Message>instance={{AMPS_INSTANCE_NAME}}</Message>
      </Options>
    </Do>
  </Action>
```

```
</Actions>
```

AMPS actions are implemented as AMPS modules. AMPS provides the modules described in the following sections by default.

24.1. Running an Action on a Schedule

AMPS provides the `amps-action-on-schedule` module for running actions on a specified schedule.

The options provided to the module define the schedule on which AMPS will run the actions in the Do element.

Table 24.2. Parameters for Scheduling Actions

Parameter	Description
`Every`	Specifies a recurring action that runs whenever the time matches the provided specification. Specifications can take three forms: • *Timer action*. A specification that is simply a duration, such as `4h` or `1d`, creates a timer action. AMPS starts the timer when the instance starts. When the timer expires, AMPS runs the action and resets the timer. • *Daily action*. A specification that is a time of day, such as `00:30` or `17:45`, creates a daily action. AMPS runs the action every day at the specified time. AMPS uses a 24 hour notation for daily actions. • *Weekly action*. A specification that includes a day of the week and a time, such as `Saturday at 11:00` or `Wednesday at 03:30` creates a weekly action. AMPS runs the action each week on the day specified, at the time specified. AMPS uses a 24 hour notation for weekly actions. AMPS accepts both local time and UTC for time specifications. To use UTC, append a `Z` to the time specifier. For example, the time specification `11:30` is 11:30 AM local time. The time specification `11:30Z` is 11:30 AM UTC.
`Name`	The name of the schedule. This name appears in log messages related to this schedule. Default: `unknown`

This module does not add any variables to the AMPS context.

24.2. Running an Action in Response to a Signal

AMPS provides the `amps-action-on-signal` module for running actions when AMPS receives a specified signal.

The module requires the `Signal` parameter:

Table 24.3. Parameters for Responding to Signals

Parameter	Description
`Signal`	Specifies the signal to respond to. This module supports the standard Linux signals. Configuring an action uses the standard name of the signal.

Parameter	Description
	For example, to configure an action to `SIGUSR1`, the value for the `Signal` element is `SIGUSR1`. To configure an action for `SIGHUP`, the value for the `Signal` element is `SIGHUP` and so on. AMPS reserves `SIGQUIT` for producing minidumps, and does not allow this module to override `SIGQUIT`. AMPS registers actions for several signals by default. See the section called “Default Signal Actions” for details.

This module does not add any variables to the AMPS context.

> ⚠ Actions can be used to override the default signal behavior for AMPS.

Default Signal Actions

By default, AMPS registers the following actions for signals.

Table 24.4. Default Actions

On Event	Action
`SIGUSR1`	`amps-action-do-disable-authentication`
`SIGUSR1`	`amps-action-do-disable-entititlement`
`SIGUSR2`	`amps-action-do-enable-authentication`
`SIGUSR2`	`amps-action-do-enable-entitlement`
`SIGINT`	`amps-action-do-shutdown`
`SIGTERM`	`amps-action-do-shutdown`
`SIGHUP`	`amps-action-do-shutdown`

The actions in the table above can be be overriden by creating an explicit action in the configuration file.

AMPS reserves `SIGQUIT` to perform the action `amps-action-do-minidump`. This behavior is reserved, and cannot be overriden.

24.3. Running an Action on Startup or Shutdown

AMPS includes modules to run actions when AMPS starts up or shuts down.

The `amps-action-on-startup` module runs actions as the last step in the startup sequence. The `amps-action-on-shutdown` module runs actions as the first step in the AMPS shutdown sequence.

In both cases, actions run in the order that the actions appear in the configuration file.

These modules do not require any parameters.

These modules do not add any variables to the AMPS context.

24.4. Running an Action on Client Connection

AMPS provides modules for running actions on the connection or disconnection of an AMPS client.

The `amps-action-on-disconnect-client` runs actions once an AMPS client instance disconnects. The `amps-action-on-connect-client` runs actions once an instance of an AMPS client successfully connects.

These modules do not require any parameters.

These modules add the following variables to the AMPS context.

Table 24.5. Context Variables for On Connect and Disconnect Client

Variable	Description
`AMPS_CLIENT_NAME`	The name of the AMPS client.
`AMPS_CONNECTION_NAME`	The name of the AMPS connection.

24.5. Running an Action on Message Delivery

AMPS provides modules to run actions when AMPS delivers a message to subscribers. The basic flow of AMPS messaging is to first receive a published message, find the subscriber(s) to which this message will be sent, then deliver the message.

The `amps-action-on-deliver-message` runs actions when AMPS delivers a message to subscribers.

This modules requires the `MessageType` and the `Topic` of the message that has been delivered:

Table 24.6. Parameters for On Deliver Message

Parameter	Description
`MessageType`	The message type of the topic to monitor for message delivery. There is no default for this parameter.
`Topic`	The name of the topic to monitor for message delivery. This parameter supports regular expressions. There is no default for this parameter.

This module adds the following variables to the AMPS context:

Table 24.7. Context Variables for On Deliver Message

Variable	Description
`AMPS_TOPIC`	The topic of the message.
`AMPS_DATA`	The data the message contains.
`AMPS_DATA_LENGTH`	The length of the data the message contains.
`AMPS_BOOKMARK`	The bookmark associated with this message. This is an empty string if the message does not have a bookmark.
`AMPS_CLIENT_NAME`	The name of the client to which this message was delivered.

24.6. Running an Action on Message Publish

AMPS provides modules to run actions when a message is published to AMPS. The basic flow of AMPS messaging is to first receive a published message, find the subscriber(s) to which this message will be sent, then deliver that message to the subscriber(s).

The `amps-action-on-publish-message` runs actions as soon as a message is published to AMPS.

This module requires the `MessageType` and the `Topic` of the message that was published. In addition to that, this module also accepts an optional `MessageSource` parameter:

Table 24.8. Parameters for On Publish Message

`MessageType`	The message type of the topic to monitor for publishes. There is no default for this parameter.
`Topic`	The name of the topic to monitor for publishes. This parameter supports regular expressions. There is no default for this parameter.
`MessageSource`	The source to monitor for publishes. The source of the message defaults to `all`, which monitors both publishes directly to this AMPS instance and messages received via replication. This parameter also accepts `local` for when the message source is published directly to this AMPS instance and `replicated` for messages received via replication.
`Filter`	Sets the filter to apply. Only messages that match this filter will cause the action to run.

This module adds the following variables to the AMPS context:

Table 24.9. Context Variables for On Publish Message

Variable	Description
`AMPS_TOPIC`	The topic of the message.
`AMPS_DATA`	The data the message contains.
`AMPS_DATA_LENGTH`	The length of the data that the message contains.
`AMPS_BOOKMARK`	The bookmark associated with this message.
`AMPS_TIMESTAMP`	The time at which the message was processed by AMPS.
`AMPS_CLIENT_NAME`	The name of the client from which the message was published.

24.7. Running an Action on OOF Message

When a record that previously matched a subscription has been updated so that the record no longer matches its subscription, AMPS sends an out-of-focus (OOF) message to let subscribers know that their record no longer matches the subscription. With `amps-action-on-oof-message`, you can enter a subscription within AMPS and run actions when an OOF message for that subscription is produced.

This module requires the following parameters:

Table 24.10. Parameters for On OOF Message

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>MessageType</td><td>The message type of the topic to monitor for OOF messages. This parameter supports regular expressions. There is no default for this parameter.</td></tr>
<tr><td>Topic</td><td>The topic to monitor for OOF messages.The topic specified must be a SOW topic, view, or conflated topic. This parameter supports regular expressions. There is no default for this parameter.</td></tr>
<tr><td>Filter</td><td>Set the filter to apply. This filter forms the internal subscription for which OOF messages will be generated.</td></tr>
<tr><td>Type</td><td>The type of OOF message to take action on.

Table 24.11. OOF message types for amps-action-on-oof-message
<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>match</td><td>Take action on OOF messages generated because message no longer matches filter.</td></tr>
<tr><td>delete</td><td>Take action on OOF messages generated because message has been removed from the SOW.</td></tr>
<tr><td>expire</td><td>Take action on OOF messages generated because the message expired from the SOW.</td></tr>
<tr><td>all</td><td>Take action on all of the above types.</td></tr>
</table>

Defaults to all.</td></tr>
</table>

This module adds the following variables to the AMPS context:

Table 24.12. Context Variables for On OOF Message

Variable	Description
AMPS_TOPIC	The topic of the OOF message.
AMPS_DATA	The data of the OOF message.
AMPS_DATA_LENGTH	The length of the data of the OOF message.
AMPS_PREVIOUS_DATA	The data previously contained from the updated record.
AMPS_PREVIOUS_DATA_LENGTH	The length of the data previously contained from the updated record.

24.8. Running an Action on Minidump

AMPS provides the `amps-action-on-minidump` module for running actions when AMPS generates a minidump.

This module does not require parameters.

This module adds the following variable to the AMPS context:

Table 24.13. Context Variable for On Minidump

Variable	Description
AMPS_MINIDUMP_PATH	The path to where the minidump is created.

24.9. Running an Action on Offline Start or Stop

AMPS provides modules to run actions when an AMPS client is marked as a slow client, and also for when the AMPS client catches up to no longer be subject to slow client offlining.

Slow client offlining is a feature in AMPS that reduces the memory resources consumed by slow clients. More on this feature can be found in the section called “Slow Client Management ”.

The `amps-action-on-offline-start` module runs actions as the first step when AMPS's result set reaches its disk limit and has to disconnect the client. The `amps-action-on-offline-stop` module runs actions as AMPS is no longer subject to slow client offlining.

In both cases, actions run in the order that the actions appear in the configuration file.

Both modules do not require any parameters.

Both modules add the following variables to the AMPS context:

Table 24.14. Context Variables for On Offline Start and Stop

Variable	Description
AMPS_CLIENT_NAME	The name of the AMPS client.
AMPS_CONNECTION_NAME	The name of the AMPS connection.

24.10. Rotate Log Files

AMPS provides the following module for rotating log files. AMPS loads this module by default:

Table 24.15. Managing Logs

Module Name	Does
amps-action-do-rotate-logs	Rotates logs that are older than a specified age, for log types that support log rotation. Rotating a log involves closing the log and opening the next log in sequence. AMPS will use the name specifier provided in the AMPS configuration for the new log file. This may overwrite the current log file if the specifier results in the same name as the current log file.

This module does not require options.

This module does not add any variables to the AMPS context:

24.11. Manage the Statistics Database

AMPS provides the following modules for managing the statistics database. As a maintenance strategy, 60East recommends truncating statistics on a regular basis. This frees space in the database file, which will be reused as new statistics are generated. It is generally not necessary to vacuum statistics unless you have changed your retention policy so that less data is retained between truncation operations. With regular truncation, the statistics database file will usually stabilize at the correct size to hold the amount of data your application generates between truncation operations.

AMPS loads these modules by default.

Table 24.16. Managing Logs

Module Name	Does
`amps-action-do-truncate-statistics`	Removes statistics that are older than a specified age. This frees space in the statistics file, but does not reduce the size of the file.
`amps-action-do-vacuum-statistics`	Remove unused space in the statistics file to reduce the size of the file. In general, it is not necessary to remove unused space in the statistics file. This operation can be expensive, and query access to the statistics database can be unavailable for an extended period of time if the file is large. If storage space is in high demand, and the interval at which the file is vacuumed has been reduced, removing space from the file can sometimes reduce the space needs. 60East recommends using this action only in long-running AMPS environments where space is at a premium, and scheduling the action during times when it is acceptable for monitoring of the system to be unavailable while the file is processed.

The `amps-action-do-truncate-statistics` module requires an `Age` parameter that specifies the age of the statistics to process.

Table 24.17. Parameters for Managing Statistics

Parameter	Description
`Age`	Specifies the age of the statistics to remove. The module processes any file older than the specified `Age`. For example, when the `Age` is `5d`, the module removes statistics that are older than 5d. There is no default for this parameter.

These modules do not add any variables to the AMPS context.

24.12. Manage Journal Files

AMPS provides the following modules for managing journal files. AMPS loads these modules by default:

Table 24.18. Managing Journals

Module Name	Does
`amps-action-do-archive-journal`	Archives journal files that are older than a specified age to the `JournalArchiveDirectory` specified for the transaction log.
`amps-action-do-compress-journal`	Compresses journal files that are older than a specified age.
`amps-action-do-remove-journal`	Deletes journal files that are older than a specified age.

Each of these modules requires an `Age` parameter that specifies the age of the journal files to process.

Table 24.19. Parameters for Managing Journals

Parameter	Description
`Age`	Specifies the age of files to process. The module processes any file older than the specified `Age`. For example, when the `Age` is `5d`, only files that have not been written to for longer than 5 days will be processed by the module. AMPS does not process the current log file, or files that are being used for replay, files that are being used for replication, or files that contain unacknowledged and unexpired messages in a queue; even if the file has been inactive for longer than the `Age` parameter. AMPS does not allow gaps in the journal files, so it will only remove a given file if all previous files have been removed. There is no default for this parameter.

These modules do not add any variables to the AMPS context.

24.13. Removing Files

AMPS provides the following module for removing files. Use this action to remove error log files that are no longer needed. AMPS loads this module by default. This action cannot be used to safely remove journal files (also known as transaction log files). For those files, use the journal management actions described in Section 24.12.

This action removes files that match an arbitrary pattern. If the pattern is not specified carefully, this action can remove files that contain important data, are required for AMPS, or are required by the operating system.

This action cannot be used to safely remove journal files. Use the actions in Section 24.12 to manage journal files.

Table 24.20. Removing Files

Module Name	Does
`amps-action-do-remove-files`	Removes files that match the specified pattern that are older than the specified age. This action accepts an arbitrary pattern, and removes files that match that pattern. While AMPS attempts to protect against deleting journal files, using a pattern that removes files that are critical for AMPS, for the application, or for the operating system may result in loss of data. The module does not recurse into directories. It skips open files. The module does not remove AMPS journals (that is, files that end with a `.journal`

Module Name	Does
	extension), and reports an error if a file with that extension matches the specified `Pattern`. The commands to remove files are executed with the current permissions of the AMPS process.

This module requires an `Age` parameter that specifies the age of the files to remove, as determined by the update to the file. This module also requires a `Pattern` parameter that specifies a pattern for locating files to remove.

Table 24.21. Parameters for Removing Files

Parameter	Description
`Age`	Specifies the age of files to process. The module removes any file older than the specified `Age` that matches the specified `Pattern`. For example, when the `Age` is `5d`, only files that have not modified within 5 days and that match the pattern will be processed by the module. There is no default for this parameter.
`Pattern`	Specifies the pattern for files to remove. The module removes any files that match the specified `Pattern` that have not been modified more recently than the specified `Age`. This parameter is interpreted as a Unix shell globbing pattern. It is *not* interpreted as a regular expression. As with other parameters that use the file system, when the pattern specified is a relative path the parameter is interpreted relative to the current working directory of the AMPS process. When the pattern specified is an absolute path, AMPS uses the absolute path. There is no default for this parameter.

This module does not add any variables to the AMPS context.

24.14. Deleting Messages from SOW

AMPS also provides modules for deleting SOW contents. The `amps-action-do-delete-sow` module deletes messages from the specified SOW topic.

This module requires the `MessageType`, `Topic`, and `Filter` parameters in order to delete the desired message.

Table 24.22. Parameters for Deleting SOW Messages

Parameter	Description
`MessageType`	The MessageType of the SOW topic or topics to delete from. There is no default for this parameter.
`Topic`	The name of the SOW topic from which to delete messages. This parameter supports regular expressions. There is no default for this parameter.

Parameter	Description
`Filter`	Set the filter to apply. Only messages matching that filter will be deleted.

These modules do not add any variables to the AMPS context.

24.15. Querying a SOW Topic

AMPS provides a module for querying a SOW topic. The `amps-action-do-query-sow` queries the SOW topic, and stores the first message returned by the SOW query into a user-defined variable.

This module requires the `MessageType`, `Topic`, and `Filter` parameters to identify the query to run. This module requires the `CaptureData` parameter in order to be able to store the result of the query.

Table 24.23. Parameters for Querying SOW Messages

Parameter	Description
`MessageType`	The message type of the topic to query. There is no default for this parameter
`Topic`	The name of the topic to query. This topic must be a SOW topic, a view, a queue, or a conflated topic. There is no default for this parameter. This parameter supports regular expressions.
`Filter`	Set the filter to apply. If a `Filter` is present, only messages matching that filter will be returned by the query.
`CaptureData`	Sets the name of the variable within which AMPS will store the first message returned.
`DefaultData`	If no records are found, AMPS stores the `DefaultData` in the variable specified by `CaptureData`.
`OrderBy`	An `OrderBy` expression to use to order the results returned by the query. For example, to order in descending order of the `/date` field in the messages, you would provide an `OrderBy` option of `/date DESC`.

Once you query messages from the SOW topic, you can use the captured data in other actions. The example below uses `amps-action-do-query-sow` to query the SOW on a schedule in order to echo messages to the log for diagnostic purposes:

```
<Actions>
  <Action>
    <On>
      <Module>amps-action-on-schedule</Module>
      <Options>
        <Every>Saturday at 23:59</Every>
        <Name>Diagnostic_Schedule</Name>
      </Options>
    </On>
    <Do>
      <Module>amps-action-do-query-sow</Module>
      <Options>
        <MessageType>xml</MessageType>
        <Topic>SOW_TOPIC</Topic>
        <Filter>/Trans/Order/@Oname = 'PURCHASE'</Filter>
        <CaptureData>AMPS_DATA</CaptureData>
```

```
      </Options>
    </Do>
    <Do>
      <Module>amps-action-do-extract-values</Module>
      <Options>
        <MessageType>xml</MessageType>
        <Data>{{AMPS_DATA}}</Data>
        <Value>VALUE = /VALUE</Value>
      </Options>
    <Do>
      <Module>amps-action-do-echo-message</Module>
      <Options>
        <Message>{{VALUE}} was in the message</Message>
      </Options>
    </Do>
  </Action>
</Actions>
```

24.16. Manage Security

AMPS provides modules for managing the security features of an instance.

Authentication and entitlement can be enabled or disabled, which is useful for debugging or auditing purposes. You can also reset security and authentication, which clears the AMPS internal caches and gives security and authentication modules the opportunity to reinitialize themselves, for example, by re-parsing an entitlements file.

AMPS loads the following modules by default:

Table 24.24. Security Modules

Module Name	Does
`amps-action-do-disable-authentication`	Disables authentication for the instance.
`amps-action-do-disable-entitlement`	Disables entitlement for the instance.
`amps-action-do-enable-authentication`	Enables authentication for the instance.
`amps-action-do-enable-entitlement`	Enables entitlement for the instance.
`amps-action-do-reset-authentication`	Resets authentication by clearing AMPS caches and reinitializing authentication
`amps-action-do-reset-entitlement`	Resets entitlement by clearing AMPS caches and reinitializing entitlement

These modules require no parameters.

These modules do not add any variables to the AMPS context.

24.17. Enable and Disable a Transport

AMPS provides modules that can enable and disable specific transports. The `amps-action-do-enable-transport` module enables a transport. The `amps-action-do-disable-transport` module disables a transport.

Table 24.25. Transport Action Modules

Module Name	Does
amps-action-do-enable-transport	Enables a specific transport.
amps-action-do-disable-transport	Disables a specific transport.

Both modules require the name of the transport to disable or enable.

Table 24.26. Parameters for Managing Transports

Parameter	Description
Transport	The Name of the transport to enable or disable. If no Name is provided, the module affects all transports.

Both modules do not add any variables to the AMPS context.

24.18. Publishing Messages

The amps-action-do-publish-message module publishes a message into a specified topic.

To publish a message, this module requires the MessageType, a Topic to publish on, and also the Data that the message will contain.

Table 24.27. Parameters for Publishing Messages

Parameter	Description
MessageType	The MessageType for the topic. There is no default for this parameter.
Topic	The topic of the message being published.
Data	The data that the message will contain.
Delta	Whether to use a delta publish. When this option is present, and the value is true, the action will use a delta publish.

This module does not add any variables to the AMPS context.

24.19. Manage Replication

AMPS provides modules for downgrading replication destinations that fall behind and upgrading them again when they catch up.

Table 24.28. Replication Modules

Module Name	Does
amps-action-do-downgrade-replication	Downgrades replication connections from synchronous to asynchronous if the age of the last acknowledged message is older than a specified time period.
amps-action-do-upgrade-replication	Upgrades previously-downgraded replication connections from asynchronous to synchronous if the age of the last acknowledged message is more recent than a specified time period. This action has no effect on replication destinations that are specified as async in the configuration file.

The modules determine when to downgrade and upgrade based on the age of the oldest message that a destination has not yet acknowledged. When using these modules, it is important that the thresholds for the modules are not set too close together. Otherwise, AMPS may repeatedly upgrade and downgrade the connection when the destination is consistently acknowledging messages at a rate close to the threshold values. To avoid this, 60East recommends that the `Age` set for the upgrade module is 1/2 of the age used for the downgrade module.

The `amps-action-do-downgrade-replication` module accepts the following options:

Table 24.29. Parameters for Downgrading Replication

Parameter	Description
`Age`	Specifies the maximum message age at which AMPS downgrades a replication destination to `async`. When this action runs, AMPS downgrades any destination for which the oldest unacknowledge message is older than the specified `Age`. For example, when the `Age` is `5m`, AMPS will downgrade any destination where a message older than 5 minutes has not been acknowledged. There is no default for this parameter.
`GracePeriod`	The approximate time to wait after start up before beginning to check whether to downgrade links. The `GracePeriod` allows time for other AMPS instances to start up, and for connections to be established between AMPS instances.

The `amps-action-do-upgrade-replication` module only applies to destinations configured as `sync` that have been previously downgraded. The module accepts the following options:

Table 24.30. Parameters for Upgrading Replication

Parameter	Description
`Age`	Specifies the maximum message age at which a previously-downgraded destination will be upgraded to `sync` mode. When this action runs, AMPS upgrades any destination that has been previously downgraded where the oldest unacknowledged message to AMPS is more recent than time value specified in the `Age` parameter. For example, if a destination has been downgraded to `async` mode and the `Age` is `2m`, AMPS will upgrade the destination when the oldest unacknowledged message to that destination is less than 2 minutes old. There is no default for this parameter.
`GracePeriod`	The approximate time to wait after start up before beginning to check whether to upgrade links. The `GracePeriod` allows time for other AMPS instances to start up, and for connections to be established between AMPS instances.

These modules do not add any variables to the AMPS context.

24.20. Extract Values

The `amps-action-do-extract-values` module extracts message values from a message and stores the values in a variable.

To extract values from a message, this module requires the `MessageType`, `Data`, and `Value` parameters.

Table 24.31. Parameters for Extract Values

Parameter	Description
`MessageType`	The MessageType for the topic. There is no default for this parameter.
`Data`	Contains the data to parse: typically a message recieved from a publish event or retrieved from a SOW query. There is no default value for this parameter. If it is omitted, AMPS will not parse data when the action is run.
`Value`	An assignment statement that specifies the variable to store the extracted value in and the XPath identifier for the value to extract. This action can contain any number of `Value` elements, each providing an assignment statement. For example, the following assigment statement stores the value of the `/previousRegionCode` within the message to the variable `PREVIOUS_REGION`. `PREVIOUS_REGION = /previousRegionCode` There is no default for this option. If no `Value` options are provided, AMPS does not save any values from the parsed message.

The module `amps-action-do-extract-values` adds the variables specified by the `Value` options to the current context.

24.21. Translate Data

The `amps-action-do-translate-data` action allows you to translate the value from variables in the current context. One common use for this action is to translate a large number of status values into a smaller number of states before publishing that information in a message. For example, an order processing system may track a large number of finely-grained status codes, while the reporting view for customers may want to map those status codes to a smaller set of codes such as "pending", "shipped", and "delivered". This action allows you to easily translate those codes within AMPS.

When used to assemble a message, this action provides equivalent results to a set of nested conditional statements in a view projection. However, if you are using actions to parse, assemble, and publish messages, this action gives you the ability to change values.

Table 24.32. Parameters for Translate Data

Parameter	Description
`Data`	The data to translate. Most often, this is the value of a variable in the current context.
`Value`	The variable to store the translated value in.
`Case`	An translation statement. The translation statement takes the form of ***`original_value`***`=`***`translated_value`***. This action allows you to provide any number of `Case` statements. The action matches the `Data` provided to the ***`original_value`*** in each `Case` statement. When it finds a matching value, the action stores the translated value in the variable identified by the `Value` statement. For example, the following translation statement translates the value `credit_check_in_progress=pending`

Parameter	Description
	There is no default for this option.
`Default`	The default translation. AMPS sets the value of the variable to the contents of this element if no Case statement matches the Data provided. This element is optional. If no `Default` is specified, AMPS uses the value of the original `Data` as the default translation.

24.22. Increment Counter

The `amps-action-do-increment-counter` module allows AMPS to increment a counter by a value. Counters persist across action runs, and are saved in the instance memory until the instance is restarted.

If a counter with the specified name does not currently exist in the instance when the action runs, AMPS creates the counter with a value of 0 and then immediately increments it with the specified value. If the counter is already present, AMPS will simply increment the counter.

To see an example of `amps-action-do-increment-counter`, refer to the Action Configuration Examples section at the end of this chapter.

This module requires a `Key` that tells AMPS which counter to increment and a `Value` that tells AMPS where to store the incremented value.

Table 24.33. Parameter for Increment Counter

Parameter	Description
`Key`	The name of the counter that AMPS will increment. There is no default value for this parameter.
`Value`	The variable in which to store the current value of the counter.

This module adds variable that contains the counter, as specified in the `Value` parameter, to the current context.

24.23. Executing System Commands

The `amps-action-do-execute-system` module allows AMPS to execute system commands.

The parameter for this module is simply the command. The command executes in the current working directory of the AMPS process, with the credentials and environment of the AMPS process.

Table 24.34. Parameter for Execute System

Parameter	Description
`Command`	The command to execute. When the action runs, this command is executed as a shell command on the system where AMPS is running.

This module does not add any variables to the AMPS context.

> ⚠ This module executes system commands with the credentials of the AMPS process. It is possible to damage the system, interrupt the AMPS service, or cause data loss by executing commands with this module. 60East recommends against using any data extracted from an AMPS message in the command executed.

24.24. Debugging AMPS

AMPS provides modules for debugging your AMPS applications.

Table 24.35. Debugging Modules

Module Name	Does
`amps-action-do-nothing`	Takes no action. Does not modify the state of AMPS in any way. The module simply logs that it was called.
`amps-action-do-echo-message`	Echoes the specified message to the log. The message appears in the log as message `29-0103`,at `info` level.The logging configuration must allow this message to be recorded for the output of this action to appear in the log.

The `amps-action-do-nothing` module requires no parameters.

The `amps-action-do-echo-message` module requires the following parameter:

Table 24.36. Parameter for Echo Message

Parameter	Descsription
`Message`	The message to echo. The default for this parameter is simply an empty string.

These modules do not add any variables to the AMPS context.

24.25. Creating a Minidump

AMPS provides a module for creating a minidumps. The `amps-action-do-minidump` module provides a way for developers and/or administrators to easily create minidumps for diagnostic purposes.

Table 24.37. Creating a Minidump Module

Module Name	Does
`amps-action-do-minidump`	Creates a minidump.

This module does not require any parameters.

This module does not add any variables to the AMPS context.

24.26. Shut Down AMPS

The `amps-action-do-shutdown` module shuts down AMPS. This module is registered as the default action for several Linux signals, as described in the section called “Default Signal Actions”.

Table 24.38. Shut Down Module

Module Name	Does
`amps-action-do-shutdown`	Shuts down AMPS.

This module does not require any parameters.

This module does not add any variables to the the AMPS context.

24.27. Action Configuration Examples

Archive Files Older Than One Week, Every Saturday

The listing below asks AMPS to archive files older than 1 week, every Saturday at 12:30 AM:

```
<Actions>
      <Action>
        <On>
          <Module>amps-action-on-schedule</Module>
          <Options>
            <Every>Saturday at 00:30</Every>
            <Name>Saturday Night Fever</Name>
          </Options>
        </On>
        <Do>
          <Module>amps-action-do-archive-journal</Module>
          <Options>
              <Age>7d</Age>
          </Options>
        </Do>
      </Action>
   </Actions>
```

Disable and Re-enable Security on Signal

The listing below disables authentication and entitlement when AMPS receives on the `USR1` signal. When AMPS receives the `USR2` signal, AMPS re-enables authentication and entitlement. This configuration is, in effect, the configuration that AMPS installs by default for these signals:

```
<Actions>
      <Action>
        <On>
          <Module>amps-action-on-signal</Module>
          <Options>
            <Signal>SIGUSR1</Signal>
          </Options>
        </On>
        <Do>
          <Module>amps-action-do-disable-authentication</Module>
        </Do>
        <Do>
          <Module>amps-action-do-disable-entitlement</Module>
        </Do>
      </Action>
```

```
      <Action>
        <On>
          <Module>amps-action-on-signal</Module>
          <Options>
            <Signal>SIGUSR2</Signal>
          </Options>
        </On>
        <Do>
          <Module>amps-action-do-enable-authentication</Module>
        </Do>
        <Do>
          <Module>amps-action-do-enable-entitlement</Module>
        </Do>
      </Action>
   </Actions>
```

Extract Values on Publish of a Message

The listing below extracts values from a locally published xml message and stores them into VALUE.

```
<Actions>
  <Action>
    <On>
      <Module>amps-action-on-publish-message</Module>
      <Options>
        <Topic>message-sow</Topic>
        <MessageType>xml</MessageType>
        <MessageSource>local</MessageSource
      </Options>
    </On>
    <Do>
      <Module>amps-action-do-extract-values</Module>
      <Options>
        <MessageType>xml</MessageType>
        <Data>{{AMPS_DATA}}</Data>
        <Value>VALUE = /VALUE</Value>
      </Options>
    </Do>
  </Action>
</Actions>
```

Increment a Counter and Echo a Message on Signal

The listing below increments a counter and echoes the counter's value when AMPS receives on the `USR1` signal.

```
<Actions>
 <Action>
   <On>
     <Module>amps-action-on-signal</Module>
     <Options>
       <Signal>SIGUSR1</Signal>
```

```
      </Options>
    </On>
    <Do>
      <Module>amps-action-do-increment-counter</Module>
      <Options>
        <Key>MY_COUNTER</Key>
        <Value>CURRENT_COUNTER_VALUE</Value>
      </Options>
    </Do>
    <Do>
      <Module>amps-action-do-echo-message</Module>
      <Options>
        <Message>AMPS has gotten {{CURRENT_COUNTER_VALUE}}
                 SIGUSR1 signals.</Message>
      </Options>
     </Do>
 </Action>
</Actions>
```

Chapter 25. Replication and High Availability

This chapter discusses the support that AMPS provides for replication, and how AMPS features help to build systems that provide high availability.

25.1. Overview of AMPS High Availability

AMPS is designed for high performance, mission-critical applications. Those systems typically need to meet availability guarantees. To reach those availability guarantees, systems need to be fault tolerant. It's not realistic to expect that networks will never fail, components will never need to be replaced, or that servers will never need maintenance. For high availability, you build applications that are fault tolerant: that keep working as designed even when part of the system fails or is taken offline for maintenance. AMPS is designed with this approach in mind. It assumes that components will occasionally fail or need maintenance, and helps you to build systems that meet their guarantees even when part of the system is offline.

When you plan for high availability, the first step is to ensure that each part of your system has the ability to continue running and deliverying correct results if any other part of the system fails. You also ensure that each part of your system can be independently restarted without affecting the other parts of the system.

The AMPS server includes the following features that help ensure high availability:

- **Transaction logging** writes messages to persistent storage. In AMPS, the transaction log is not only the definitive record of what messages have been processed, it is also fully queryable by clients. Highly available systems make use of this capability to keep a consistent view of messages for all subscribers and publishers. The AMPS transaction log is described in detail in Chapter 15.

- **Replication** allows AMPS instances to copy messages between instances. AMPS replication is peer-to-peer, and any number of AMPS instances can replicate to any number of AMPS instances. Replication can be filtered by topic. By default, AMPS instances only replicate messages published to that instance. An AMPS instance can also replicate messages received via replication using *passthrough replication*: the ability for instances to pass replication messages to other AMPS instances.

- **Heartbeat monitoring** to actively detect when a connection is lost. Each client configures the heartbeat interval for that connection.

The AMPS client libraries include the following features to help ensure high availability:

- **Heartbeat monitoring** to actively detect when a connection is lost. As mentioned above, the interval for the heatbeat is configurable on a connection-by-connection basis. The interval for heartbeat can be set by the client, allowing you to configure a longer timeout on higher latency connections or less critical operations, and a lower timeout on fast connections or for clients that must detect failover quickly.

- **Automatic reconnection and failover** allows clients to automatically reconnect when disconnection occurs, and to locate and connect to an active instance.

- **Guaranteed publication** from clients, including an optional persistent message store. This allows message publication to survive client restarts as well as server failover.

- **Subscription recovery** and **transaction log playback** allows clients to recover the state of their messaging after restarts. These features guarantee that clients recieve all messages published in the order published, including messages received while the clients were offline. These features are provided by the transaction log, as described in Chapter 15.

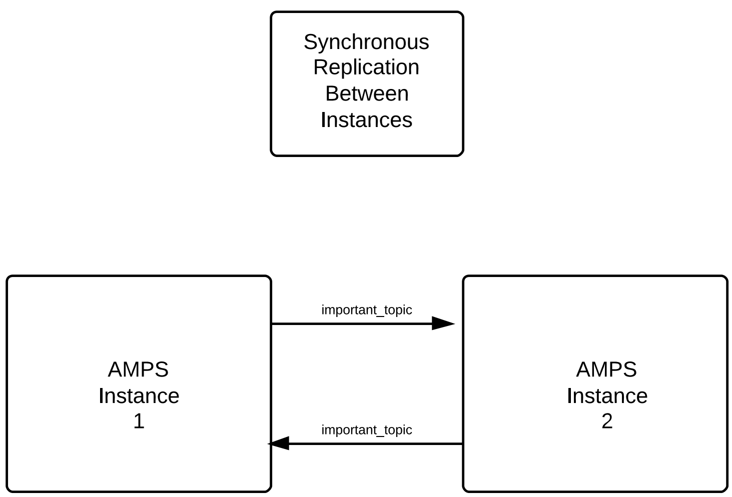
For details on each client library, see the developer's guide for that library. Further samples can be found in the evaluation kit for the client, available from the 60East website at `http://www.crankuptheamps.com/evaluate`.

25.2. High Availability Scenarios

You design your high availability strategy to meet the needs of your application, your business, and your network. This section describes commonly-deployed scenarios for high availability.

Failover Scenario

One of the most common scenarios is for two AMPS instances to replicate to each other. This replication is synchronous, so that both instances persist a message before AMPS acknowledges the message to the publisher. This makes a hot-hot pair. In the figure below, any messages published to `important_topic` are replicated across instances, so both instances have the messages for `important_topic`.

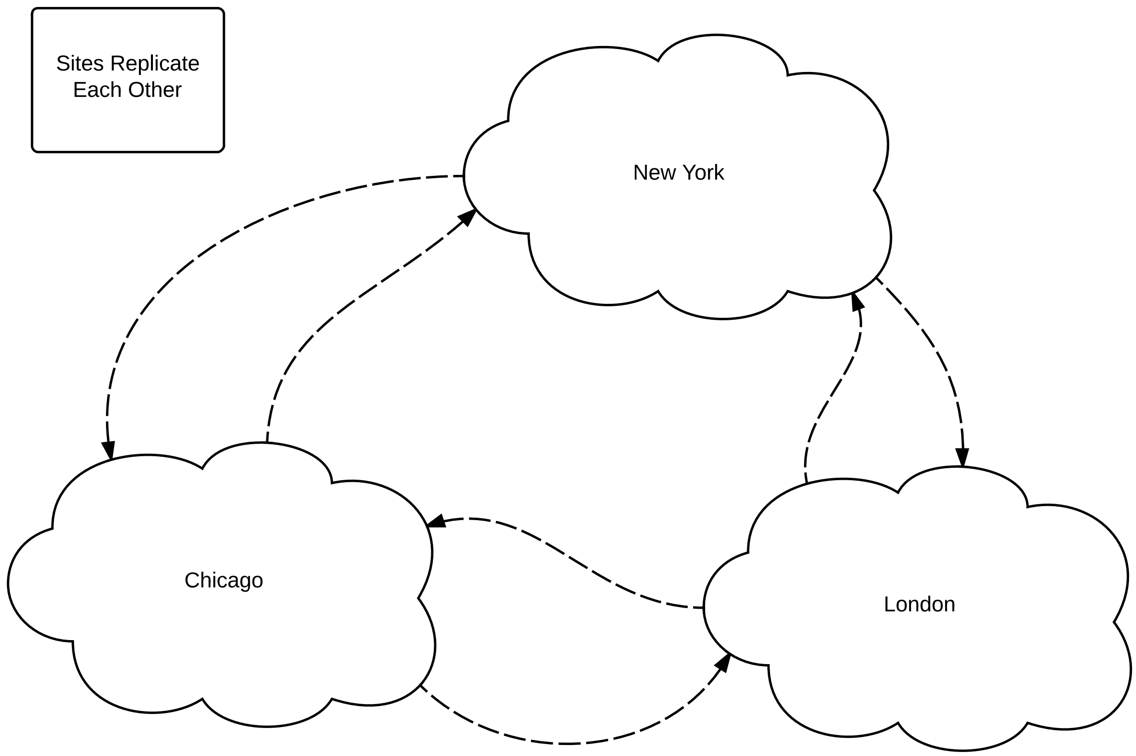


Notice that, because AMPS replication is peer-to-peer, clients can connect to either instance of AMPS when both are running. Further, messages can be published to either instance of AMPS and be replicated to the other instance. In this case, clients are configured with the addresses of both AMPS instances.

In this case, clients are configured with Instance 1 and Instance 2 as equivalent server addresses. If a client cannot connect to one instance, it tries the other. Because both instances contain the same messages for `important_topic`, there is no functional difference in which instance a client connects to. Because these instances replicate to each other, AMPS can optimize this to a single connection. Two connections are shown in the diagram to demonstrate the required configuration.

Geographic Replication

AMPS is well suited for replicating messages to different regions, so clients in those regions are able to quickly receive and publish messages to a local instance. In this case, each region replicates all messages on the topic of interest to the other two regions. A variation on this strategy is to use a region tag in the content, and use content filtering so that each replicates messages intended for use in the other regions or worldwide.



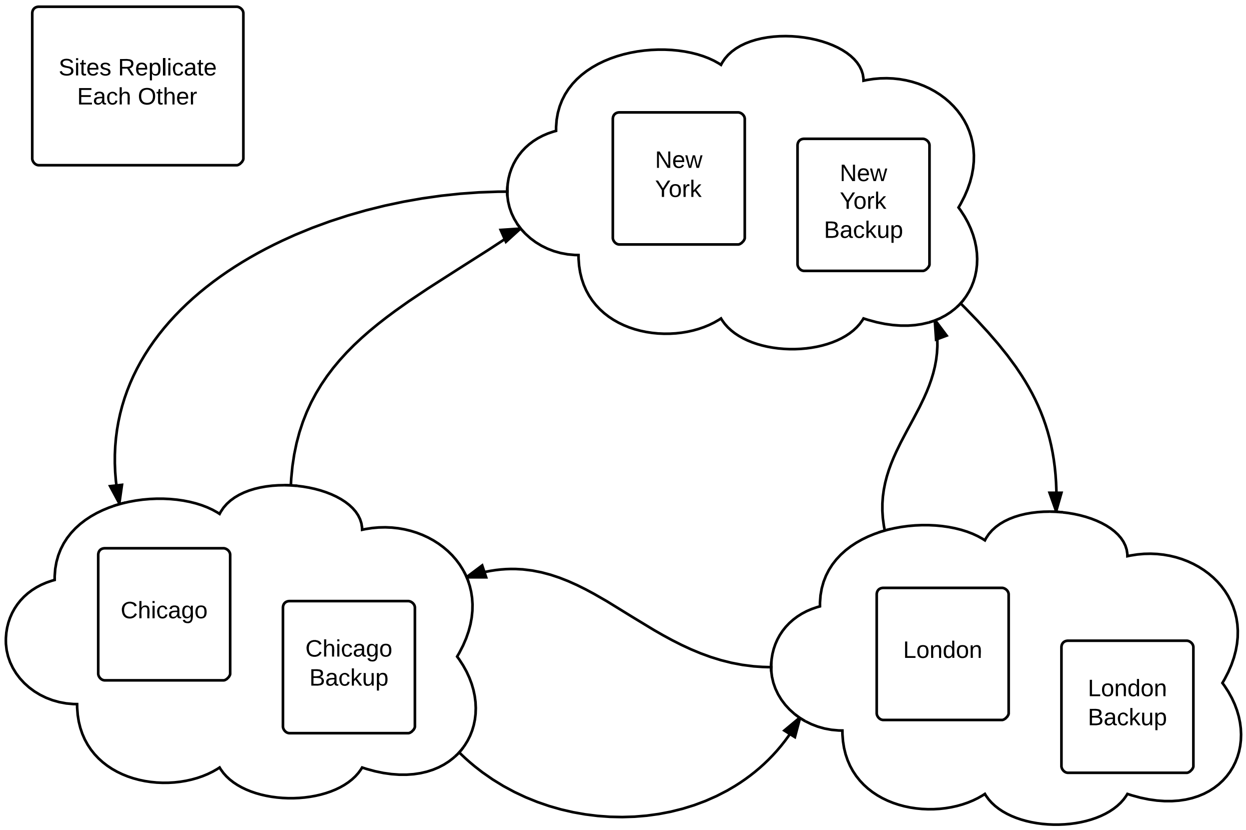
For this scenario, an AMPS instance in each region replicates to an instance in the two other regions. For the best performance, replication between the regions is asynchronous, so that once an instance in one region has persisted the message, the message is acknowledged back to the publisher.

In this case, clients in each region connect to the AMPS instance in that region. Bandwidth within regions is conserved, because each message is replicated once to the region, regardless of how many subscribers in that region will receive the message. Further, publishers are able to publish the message once to a local instance over a relatively fast network connection rather than having to publish messages multiple times to multiple regions.

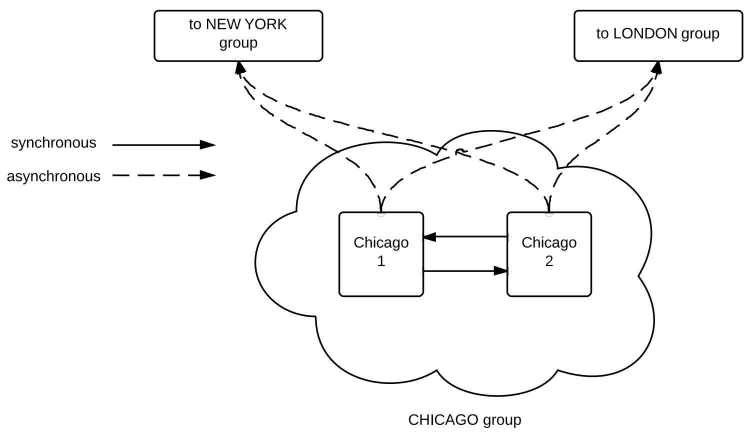
To configure this scenario, the AMPS instances in each region are configured to forward messages to known instances in the other two regions.

Geographic Replication with High Availability

Combining the first two scenarios allows your application to distribute messages as required and to have high availability in each region. This involves having two or more servers in each region, as shown in the figure below.



Each region is configured as a group. Within each group, the instances replicate to each other synchronously, and replicate to the remote instances asynchronously. The figure below shows the expanded detail of the configuration for these servers.



The instances in each region are configured to be part of a group for that region. Within a region, the instances synchronously replicate to each other, and asynchronously replicate to instances at each remote site. The instances use the replication downgrade action to ensure that message publishing continues in the event that one of the instances goes offline. As with all connections where instances replicate to each other, this replication is configured as one connection in each direction, although AMPS may optimize this to a single replication connection.

Each instance at a site provides passthrough replication from the other sites to local instances, so that once a message arrives at the site, it is replicated to the other instances at the local site. The remote sites are configured in the same way. This configuration balances fault-tolerance and performance.

Each instance at a site replicates to the remote sites. The instance specifies one `Destination` for each remote site, with the servers at the remote site listed as failover equivalents for the remote site. With the passthrough configuration, this ensures that each message is delivered to each remote site exactly once. Whichever server at the remote site receives the message distributes it to the other server using passthrough replication.

With this configuration, publishers at each site publish to the primary local AMPS instance, and subscribers subscribe to messages from their local AMPS instances. Both publishers and subscribers use the high availability features of the AMPS client libraries to ensure that if the primary local instance AMPS fails, they automatically failover to the other instance. Replication is used to deliver both high availability and disaster recovery. In the table below, each row represents a replication destination. Servers in brackets are represented as sets of `InetAddr` elements in the `Destination` definition.

Table 25.1. Geographic Replication with HA Destinations

Server	Destinations
Chicago 1	sync to Chicago 2 async to [NewYork 1, NewYork 2] async to [London 1, London 2]

Server	Destinations
Chicago 2	sync to Chicago 2 async to [NewYork 1, NewYork 2] async to [London 1, London 2]
NewYork 1	sync to NewYork 2 async to [Chicago 1, Chicago 2] async to [London 1, London 2]
NewYork 2	sync to NewYork 1 async to [Chicago 1, Chicago 2] async to [London 1, London 2]
London 1	sync to London 2 async to [Chicago 1, Chicago 2] async to [NewYork 1, NewYork 2]
London 2	sync to London 1 async to [Chicago 1, Chicago 2] async to [NewYork 1, NewYork 2]

25.3. AMPS Replication

AMPS has the ability to replicate messages to downstream AMPS instances once those messages are stored to a transaction log. Replication in AMPS involves the configuration of two or more instances designed to share some or all of the published messages. Replication is an efficient way to split and share message streams between multiple sites where each downstream site may only want a subset of the messages from the upstream instances. Additionally, replication can be used to improve the availability of a set of AMPS instances by creating redundant instances for fail-over cases.

AMPS supports two forms of replication links: *synchronous* and *asynchronous*; these settings control when publishers of messages are sent `persisted` acknowledgments. These settings do not affect when or how messages are replicated, or when or how messages are delivered to subscribers. These settings only affect when AMPS acknowledges to the publisher that the message has been persisted.

AMPS replication consists of a message stream (or, more precisely, a command stream) provided to downstream instances. AMPS replicates `publish` commands and `sow_delete` commands. AMPS does not replicate messages produced internally by AMPS, such as the results of `Views` or updates sent to a `ConflatedTopic`. When replicating Queues, AMPS also uses the replication connection to send and receive administrative commands related to queues, as described in the section on Replicated Queues.

> To replicate between two instances, both instances must have the same major and minor version number of AMPS. For example, an instance running 3.5.0.5 can replicate to an instance running 3.5.0.6, but could not replicate to an instance running 3.8.0.0. .

Configuration

Replication configuration involves the configuration of two or more instances of AMPS. For testing purposes both instances of AMPS can reside on the same physical host before deployment into a production environment. When running both instances on one machine, the performance characteristics will differ from production, so running both instances on one machine is more useful for testing configuration correctness than testing overall performance.

> It's important to make sure that when running multiple AMPS instances on the same host that there are no conflicting ports. AMPS will emit an error message and will not start properly if it detects that a port is already in use.

For the purposes of explaining this example, we're going to assume a simple primary-secondary replication case where we have two instances of AMPS - the first host is named `amps-1` and the second host is named `amps-2`. Each of the instances are configured to replicate data to the other —that is to say, all messages published to `amps-1` are replicated to `amps-2` and vice versa. This configuration ensures that the data on our two instances are always synchronized in case of a failover.

We will first show the relevant portion of the configuration used in `amps-1`, and then we will show the relevant configuration for `amps-2`.

> All replication topics must also have a Transaction Log defined. The examples below omit the Transaction Log configuration for brevity. Please reference the Transaction Log chapter for information on how to configure a transaction log for a topic.

```
<AMPSConfig>
   <Name>amps-1</Name>
   <Group>DataCenter-NYC-1</Group>
...

  <Transports>
    <Transport>
      ❶<Name>amps-replication</Name>
       <Type>amps-replication</Type>
       <InetAddr>localhost:10004</InetAddr>
       <ReuseAddr>true</ReuseAddr>
    </Transport>
    <Transport>
      ❷<Name>tcp-fix</Name>
       <MessageType>fix</MessageType>
       <Type>tcp</Type>
       <InetAddr>localhost:9004</InetAddr>
       <Protocol>fix</Protocol>
       <ReuseAddr>true</ReuseAddr>
    </Transport>
  </Transports>

...

 ❸<Replication>
    ❹<Destination>
       ❺<Topic>
           <MessageType>fix</MessageType>
```

```
          ❻<Name>orders</Name>
          ❼<Filter>/55='IBM'</Filter>
        </Topic>
        <Name>amps-2</Name>
       ❽<Group>DataCenter-NYC-1</Group>
       ❾<SyncType>sync</SyncType>
       ❿<Transport>
           ⓫<InetAddr>amps-2-server.example.com:10005</InetAddr>
            <Type>amps-replication</Type>
        </Transport>
     </Destination>
  </Replication>

...

</AMPSConfig>
```

Example 25.1. Configuration used for amps-1

❶ The amps-replication transport is required. This is a proprietary message format used by AMPS to replicate messages between instances. This AMPS instance will receive replication messages on this transport. The instance can receive messages from any number of upstream instances on this transport.

❷ The fix transport defines the message transport on port 9004 to use the FIX message type. All messages sent to this port will be parsed as FIX messages.

❸ All replication destinations are defined inside the Replication block.

❹ Each individual replication destination requires a Destination block.

❺ The replicated topics and their respective message types are defined here. AMPS allows any number of Topic definitions in a Destination.

❻ The Name definition specifies the name of the topic or topics to be replicated. The Name option can be either a specific topic name or a regular expression that matches a set of topic names.

When a specific topic is specified, that topic must be recorded in a transaction log. When a regular expression is specified, only topics of the same message type that are recorded in a transaction log are replicated.

❼ This Topic definition uses a filter that matches only when the FIX tag 55 matches the string 'IBM'. This means that messages that match only messages in topic orders with ticker symbol (tag 55) of IBM will be sent to the downstream replica amps-2.

The Topic/Filter option supports any valid AMPS filter expression. This filtering provides for greater control over the flow of messages to replicated instances.

❽ The group name of the destination instance (or instances). The name specified here must match the Group defined for the remote AMPS instance, or AMPS reports an error and refuses to connect to the remote instance.

❾ Replication SyncType can be either sync or async.

❿ The Transport definition defines the location to which this AMPS instance will replicate messages. The InetAddr points to the hostname and port of the downstream replication instance. The Type for a replication instance should always be amps-replication.

⓫ The address, or list of addresses, for the replication destination.

For the configuration amps-2, we will use the following in Example 25.2. While this example is similar, only the differences between the amps-1 configuration will be called out.

```
<AMPSConfig>
  <Name>amps-2</Name>
```

```
   <Group>DataCenter-NYC-1</Group>
...

 ❶<Transports>
    <Transport>
       <Name>amps-replication</Name>
       <Type>amps-replication</Type>
      ❷<InetAddr>10005</InetAddr>
       <ReuseAddr>true</ReuseAddr>
    </Transport>
    <Transport>
       <Name>tcp-fix</Name>
       <Type>fix</Type>
       <InetAddr>localhost:9005</InetAddr>
       <ReuseAddr>true</ReuseAddr>
    </Transport>
  </Transports>

...

  <Replication>
     <Destination>
        <Topic>
           <MessageType>fix</MessageType>
           <Name>topic</Name>
        </Topic>
        <Name>amps-1</Name>
        <Group>DataCenter-NYC-1</Group>
       ❸<SyncType>async</SyncType>
        <Transport>
          ❹<InetAddr>amps-1-server.example.com:10004</InetAddr>
           <Type>amps-replication</Type>
        </Transport>
     </Destination>
  </Replication>

...

 </AMPSConfig>
```

Example 25.2. Configuration used for amps-2

❶ The `amps-replication` transport is required. This is a proprietary message format used by AMPS to replicate messages between instances. This AMPS instance will receive replication messages on this transport. The instance can receive messages from any number of upstream instances on this transport.

❷ The port where `amps-2` listens for replication messages matches the port where `amps-1` is configured to send its replication messages. This AMPS instance will receive replication messages on this transport. The instance can receive messages from any number of upstream instances on this transport.

❸ The `amps-2` instance is configured to use a `async` for the replication destination's `SyncType`. A detailed explanation of the difference between the `sync` and `async` options for the `SyncType` can be found here: the section called “Sync vs Async”.

❹ The replication destination port for `amps-2` is configured to send replication messages to the same port on which `amps-1` is configured to listen for them.

Automatic Configuration Validation

Replication can involve coordinating configuration among a large number of AMPS instances. It can sometimes be time consuming to ensure that all of the instances are configured correctly, and to ensure that a configuration change for one instance is also made at the replication destinations. For example, if a high-availability pair replicates the topics ORDERS, INVENTORY, and CUSTOMERS to a remote disaster recovery site, but the disaster recovery site only replicates ORDERS and INVENTORY back to the high-availability pair, disaster recovery may not occur as planned. Likewise, if only one member of the HA pair replicates ORDERS to the other member of the pair, the two instances will contain different messages, which could cause problems for the system.

Starting in the 5.0 release, AMPS automatic replication configuration validation makes it easier to keep configuration items consistent across a replication fabric.

Automatic configuration validation is enabled by default. You can turn off validation for specific elements of the configuration, including turning off validation for the topic altogether by excluding all of the checks. When validation is enabled, AMPS verifies the configuration of a remote instance when a replication connection is made. If the configuration is not compatible with the source for one or more of the validation checks, AMPS logs the incompatible configuration items and does not allow the connection.

Each `Topic` in a replication `Destination` can configure a unique set of validation checks. By default, all of the checks apply to all topics in the `Destination`.

The table below lists the elements that AMPS validates:

Table 25.2. Replication Configuration Validation

Check	Validates
`txlog`	The topic is contained in the transaction log of the remote instance.
`replicate`	The topic is replicated from the remote instance back to this instance.
`sow`	If the topic is a SOW topic in this instance, it must also be a SOW topic in the remote instance.
`cascade`	The remote instance must enforce the same set of validation checks for this topic as this instance does.
`queue`	If the topic is a queue in this instance, it must also be a queue in the remote instance. This option cannot be excluded.
`keys`	If the topic is a SOW topic in this instance, it must also be a SOW topic in the remote instance and the SOW in the remote instance must use the same `Key` definitions.
`replicate_filter`	If this topic uses a replication filter, the remote instance must use the same replication filter for replication back to this instance.
`queue_passthrough`	If the topic is a queue in this instance, the remote instance must support passthrough from this group. This option cannot be excluded.
`queue_underlying`	If the topic is a queue in this instance, it must use the same underlying topic definition and filters in the remote instance.

Check	Validates
	This option cannot be excluded.

For example, the following Topic does not require the remote destination to replicate back to this instance, and does not require that the remote destination enforce the same configuration checks for any downstream replication of this topic.

```
<Destination>
...
  <Topic>
      <MessageType>json</MessageType>
      <Name>MyStuff-VIEW</Name>
      <ExcludeValidation>replicate,cascade</ExcludeValidation>
  </Topic>
...
</Destination>
```

Benefits of Replication

Replication can serve two purposes in AMPS. First, it can increase the fault-tolerance of AMPS by creating a spare instance to cut over to when the primary instance fails. Second, replication can be used in message delivery to a remote site.

In order to provide fault tolerance and reliable remote site message delivery, for the best possible messaging experience, there are some guarantees and features that AMPS has implemented. Those features are discussed below.

Replication in AMPS supports filtering by both topic and by message content. This granularity in filtering allows replication sources to have complete control over what messages are sent to their downstream replication instances.

Additionally, replication can improve availability of AMPS by creating a redundant instance of an AMPS server. Using replication, all of the messages which flow into a primary instance of AMPS can be replicated to a secondary spare instance. This way, if the primary instance should become unresponsive for any reason, then the secondary AMPS instance can be swapped in to begin processing message streams and requests.

Sync vs Async

When publishing to AMPS, it is recommended that publishers request a `persisted` acknowledgment message response. The `persisted` acknowledgement message is one of the ways in which AMPS guarantees that a message received by AMPS is stored in accordance with the configuration.

Depending on how AMPS is configured, that `persisted` acknowledgment message will be delivered to the publisher at different times in the replication process. There are two options: *synchronous* or *asynchronous*. These two `SyncType` configurations control when publishers of messages are sent `persisted` acknowledgments.

In *synchronous* replication, AMPS will not return a `persisted` acknowledgment to the publisher for a message until the message has been stored to the local transaction log, to the SOW, and to all downstream synchronous replication destinations. Figure 25.1 shows the cycle of a message being published in a replicated instance, and the persisted acknowledgment message being returned back to the publisher. Notice that, with this configuration, the publisher will not recieve an acknowledgement if the remote destination is unavailable. 60East recommends that when you use `sync` replication, you also set a policy for downgrading the link when a destination is offline, as described in the section called “Automatically Downgrading an AMPS instance”.

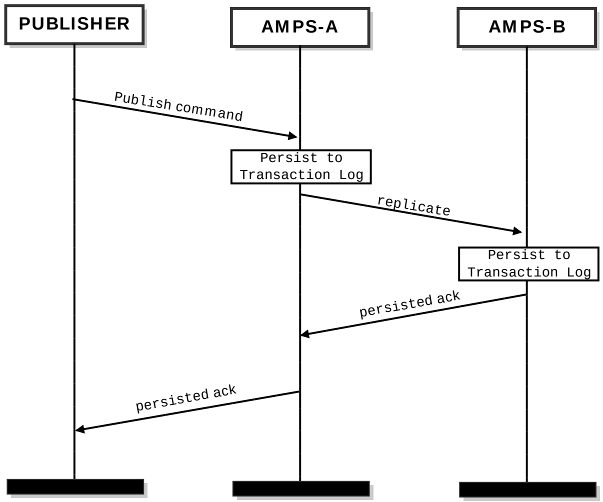


Figure 25.1. Synchronous Persistence Acknowledgment

In *asynchronous* replication, the primary AMPS instance sends the `persisted` acknowledgment message back to the publisher as soon as the message is stored in the local transaction log and SOW stores. The primary AMPS instance then sends the message to downstream replica instances. Figure 25.2 shows the cycle of a message being published with a `SyncType` configuration set to *asynchronous*.

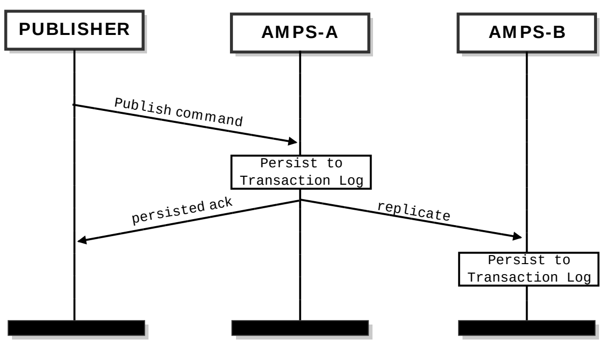


Figure 25.2. Asynchronous Persistence Acknowledgment

Replication Compression

AMPS provides the ability to compress the replication connnection. In typical use, using replication compression can greatly reduce the bandwidth required between AMPS instances.

The precise amount of compression that AMPS can achieve depends on the content of the replicated messages. Compression is configured at the replication source, and does not need to be enabled in the transport configuration at the instance receiving the replicated messages.

For AMPS instances that are receiving replicated messages, no additional configuration is necessary. AMPS automatically recognizes when an incoming replication connection uses compression.

Destination Server Failover

Your replication plan may include replication to a server that is part of a highly-available group. There are two common approaches to destination server failover.

Wide IP AMPS replication works transparently with wide IP, and many installations use wide IP for destination server failover. The advantage of this approach is that it requires no additional configuration in AMPS, and redundant servers can be added or removed from the wide IP group without reconfiguring the instances that replicate to the group. A disadvantage to this approach is that failover can require several seconds, and messages are not replicated during the time that it takes for failover to occur.

AMPS failover AMPS allows you to specify multiple downstream servers in the `InetAddr` element of a destination. In this case, AMPS treats the set list of servers as a list of equivalent servers, listed in order of priority.

When multiple addresses are specified for a destination, each time AMPS needs to make a connection to a destination, AMPS starts at the beginning of the list and attempts to connect to each address in the list. If AMPS is unable to connect to any address in the list, AMPS waits for a timeout period, then begins again with the first server on the list. Each time AMPS reaches the end of the list without establishing a connection, AMPS increases the timeout period.

This capability allows you to easily set up replication to a highly-available group. If the server you are replicating to fails over, AMPS uses the prioritized list of servers to re-establish a connection.

Back Replication

Back Replication is a term used to describe a replication scenario where there are two instances of AMPS—termed `AMPS-A` and `AMPS-B` for this example—in a special replication configuration. `AMPS-A` will be considered the primary replication instance, while `AMPS-B` will be the backup instance.

In a *back replication*, messages that are published to `AMPS-A` are replicated to `AMPS-B`. Likewise, all messages which are published to `AMPS-B` are replicated to `AMPS-A`. This replication scheme is used when both instances of AMPS need to be in sync with each other to handle a failover scenario with no loss of messages between them. This way, if `AMPS-A` should fail at any point, the `AMPS-B` instance can be brought in as the primary instance. All publishers and subscribers can quickly be migrated to the `AMPS-B` instance, allowing message flow to resume with as little downtime as possible.

In back replication, you need to decide if replication is synchronous in both directions, or synchronous from the primary, `AMPS-A`, to the secondary, `AMPS-B`, and asychronous from the secondary to the primary. If clients are actively connecting to both instances, synchronous replication in both directions provides the most consistent view of message state. If `AMPS-B` is only used for failover, then asynchronous replication from `AMPS-B` to `AMPS-A` is recommended. For any synchronous replication, consider configuring automatic replication downgrade, described below.

Starting with the 5.0 release, when AMPS detects back replication between a pair of instances, AMPS will prefer using a single connection between the servers, replicating messages in both directions over the single connection. This is particularly useful for situations where you need to have messages replicated, but only one server can initiate a connection: for example, when one of the servers is in a DMZ, and cannot make a connection to a server within the company. AMPS also allows you to specify a replication destination with no InetAddr provided: in this case, the instance will replicate once the destination establishes a destination, but will not initiate a connection. When both instances specify an InetAddr, AMPS may temporarily create two connections between the instances while replication is being established. In this case, after detecting that there are two connections active, AMPS will close one of the connections and use a single connection for replication.

Passthrough Replication

Passthrough Replication is a term used to describe the ability of an AMPS instance to pass along replicated messages to a another AMPS instance. This allows you to easily keep multiple failover or DR destinations in sync from a single AMPS instance. Unless passthrough replication is configured, an AMPS instance only replicates messages published to that instance.

Passthrough replication uses the name of the originating AMPS group to indicate that messages that arrive from that group are to be replicated to the specified destination. Passthrough replication supports regex server groups, and allows multiple server groups per destination. Notice that if the destination instance does not specify a group, the name of the instance is the name of the group.

```
<Replication>
          <Destination>
            <Name>AMPS2-HKG</Name>
            <Group>HKG</Group>
            <Transport>
                <Name>amps-replication</Name>
                <Type>amps-replication</Type>
                <InetAddr>secondaryhost:10010</InetAddr>
                <ReuseAddr>true</ReuseAddr>
            </Transport>
            <Topic>
               <Name>/rep_topic</Name>
                <MessageType>fix</MessageType>
            </Topic>
            <Topic>
                <Name>/rep_topic2</Name>
                <MessageType>fix</MessageType>
            </Topic>
            <SyncType>sync</SyncType>
            ❶<PassThrough>^NYC</PassThrough>
         </Destination>
</Replication>
```

❶ The server group from which messages will be passed through. This example passes along messages from AMPS instances from any group name that begins with `NYC`. Messages that originated at an instance that is not in a group that matches `^NYC` are not passed through to this destination. While the `PassThrough` element supports regular expressions for group names, in many cases all instances for a passthrough rule will be in the same group.

When a message is eligible for passthrough replication, topic and content filters in the replication destination still apply. The passthrough directive simply means that the message is eligible for replication from this instance if it comes from an instance in the specified group.

AMPS protects against loops in passthrough replication by tracking the instance names that a message has passed through. AMPS does not allow a message to travel through the same instance more than once.

Guarantees on ordering

For each publisher, on a single topic, AMPS is guaranteed to deliver messages to subscribers in the same order that the messages were published by the original publisher. This guarantee holds true regardless of how many publishers or how many subscribers are connected to AMPS at any one time.

For each instance, AMPS is guaranteed to deliver messages in the order in which the messages were received by the instance, regardless of whether a message is received directly from a publisher or indirectly via replication. The message order for the instance is recorded in the transaction log, and is guaranteed to remain consistent across server restarts.

These guarantees mean that subscribers will not spend unnecessary CPU cycles checking timestamps or other message content to verify which message is the most recent, or reordering messages during playback. This frees up subscriber resources to do more important work.

AMPS preserves an absolute order across topics for a single subscription for all topics *except* views, queues, and conflated topics. Applications often rely on this behavior to correlate the times at which messages to different topics were processed by AMPS. See Section 4.5 for more information.

Automatically Downgrading an AMPS instance

The AMPS administrative console provides the ability to downgrade a replication link from *synchronous* to *asynchronous*. This feature is useful should a downstream AMPS instance prove unstable, unresponsive, or introduce additional latency.

Downgrading a replication link to *asynchronous* means that any `persisted` acknowledgment message that a publisher may be waiting on will no longer wait for the downstream instance to confirm that it has committed the message to its downstream Transaction Log or SOW store. AMPS immediately considers the downstream instance to have acknowledged the message for existing messages, which means that if AMPS was waiting for acknowledgement from that instance to deliver a `persisted` acknowledgement, AMPS immediately sends the `persisted` acknowledgement when the instance is downgraded..

AMPS can be configured to automatically downgrade a replication link to `asychronous` if the remote side of the link cannot keep up with persisting messages or becomes unresponsive. This option prevents unreliable links from holding up publishers, but increases the chances of a single instance failure resulting in message loss, as described above.

Automatic downgrade is implemented as an AMPS action. To configure automatic downgrade, add the appropriate action to the configuration file as shown below:

```
<AMPSConfig>
 ...
 <Actions>
     <Action>
       <On>
         <Module>amps-action-on-schedule</Module>
         <Options>
           ❶<Every>15s</Every>
         </Options>
       </On>
       <Do>
       <Module>amps-action-do-downgrade-replication</Module>
         <Options>
           ❷<Age>30s</Age>
         </Options>
       </Do>
     </Action>
  </Actions>
  ...
```

```
</AMPSConfig>
```

❶ This option determines how often AMPS checks whether destinations have fallen behind. In this example, AMPS checks destinations every 15 seconds. In most cases, 60East recommends setting this to half of the `Interval` setting.

❷ The maximum amount of time for a destination to fall behind. If AMPS has been waiting for an acknowledgement from the destination for longer than the `Interval`, AMPS downgrades the destination. In this example, AMPS downgrades any destination for which an acknowledgment has taken longer than 30 seconds.

In this configuration file, AMPS checks every 15 seconds to see if a destination has fallen behind by 30 seconds. This helps to guarantee that a destination will never exceed the `Interval`, even in situations where the destination begins falling behind exactly at the time AMPS checks for the destination keeping up.

Replication Security

AMPS allows authorization and entitlement to be configured on replication destinations. For the instance that receives connections, you simply configure Authentication and Entitlement for the transport definition for the destination, as shown below:

```
<Transports>
    <Transport>
       <Name>amps-replication</Name>
       <Type>amps-replication</Type>
       <InetAddr>10005</InetAddr>
       <ReuseAddr>true</ReuseAddr>
       ❶<Entitlement>
          <Module>amps-default-entitlement-module</Module>
       </Entitlement>
       ❷<Authentication>
          <Module>amps-default-authentication-module</Module>
       </Authentication>
    </Transport>
 ...
</Transports>
```

❶ Specifies the entitlement module to use to check permissions for incoming connections. The module specified must be defined in the `Modules` section of the config file, or be one of the default modules provided by AMPS. This snippet uses the default module provided by AMPS for example purposes.

❷ Specifies the authorization module to use to verify identity for incoming connections. The module specified must be defined in the `Modules` section of the config file, or be one of the default modules provided by AMPS. This snippet uses the default module provided by AMPS for example purposes.

For incoming connections, configuration is the same as for other types of transports.

For connections from AMPS to replication destinations, you can configure an Authenticator module for the destination transport. Authenticator modules provide credentials for outgoing connections from AMPS. For authentication protocols that require a challenge and response, the Authenticator module handles the responses for the instance requesting access.

```
<Replication>
     <Destination>
        <Topic>
           <MessageType>fix</MessageType>
           <Name>topic</Name>
```

```
        </Topic>
        <Name>amps-1</Name>
       <SyncType>async</SyncType>
        <Transport>
          <InetAddr>amps-1-server.example.com:10004</InetAddr>
           <Type>amps-replication</Type>
           ❶<Authenticator>
               <Module>amps-default-authenticator-module</Module>
           </Authenticator>
        </Transport>
     </Destination>
  </Replication>
```

❶ Specifies the authenticator module to use to provide credentials for the outgoing connection. The module specified must be defined in the `Modules` section of the config file, or be one of the default modules provided by AMPS. This snippet uses the default module provided by AMPS for example purposes.

Maximum downstream destinations

AMPS has support for up to 64 synchronous downstream replication instances and unlimited asynchronous destinations.

25.4. High Availability

AMPS High Availability, which includes multi-site replication and the transaction log, is designed to provide long uptimes and speedy recovery from disasters. Replication allows deployments to improve upon the already rock-solid stability of AMPS. Additionally, AMPS journaling provides the persisted state necessary to make sure that client recovery is fast, painless, and error free.

Guaranteed Publishing

An interruption in service while publishing messages could be disastrous if the publisher doesn't know which message was last persisted to AMPS. To prevent this from happening, AMPS has support for *guaranteed publishing*.

The `logon` command supports a `processed` acknowledgment message, which will return the `Sequence` of the last record that AMPS has persisted. When the `processed` acknowledgment message is returned to the publisher, the `Sequence` corresponds to the last message persisted by AMPS. The publisher can then use that sequence to determine if it needs to 1) re-publish messages that were not persisted by AMPS, or 2) continue publishing messages from where it left off. Acknowledging persisted messages across logon sessions allows AMPS to guarantee publishing. The HAClient classes in the AMPS clients manage sequence numbers, including setting a meaningful initial sequence number based on the response from the `logon` command, automatically.

> It is recommended as a best practice that all publishers request a `processed` acknowledgment message with every `logon` command. This ensures that the `Sequence` returned in the acknowledgement message matches the publisher's last published message. The 60East AMPS clients do this automatically when using the named logon methods. If you are building the command yourself or using a custom client, you may need to add this request to the command yourself.

In addition to the acknowledgment messages, AMPS also keeps track of the published messages from a client based on the client's name. When sending a `logon` command, the client should also set the `ClntName` field during the `logon`.

> All publishers must set a unique `ClntName` field as part of their logon. This allows AMPS to correlate `SeqId` fields and acknowledgment messages with a specific client in the event that they should nee to reconnect. In the event that multiple publishers have the same `ClntName`, AMPS can no longer reliably correlate messages using the `SeqId` and `ClntName`.

Durable Publication and Subscriptions

The AMPS client libraries include features to enable durable subscription and durable publication. In this chapter we've covered how publishing messages to a transaction log persists them. We've also covered how the transaction log can be queried (subscribed) with a bookmark for replay. Now, putting these two features together yields *durable subscriptions*.

A *durable subscriber* is one that receives all messages published to a topic (including a regular expression topic), even when the subscriber is offline. In AMPS this is accomplished through the use of the bookmark subscription on a client.

Implementation of a *durable subscription* in AMPS is accomplished on the client by persisting the last observed bookmark field received from a subscription. This enables a client to recover and resubscribe from the exact point in the transaction log where it left off.

A *durable publisher* maintains a persistent record of messages published until AMPS acknowledges that the message has been persisted. Implementation of a durable publisher in AMPS is accomplished on the client by persisting outgoing messages until AMPS sends a `persisted` acknowledgement that says that this message, or a later message, has been persisted. At that point, the publishers can remove the message from the persistent store. Should the publisher restart, or should AMPS fail over, the publisher can re-send messages from the persistent store. AMPS uses the sequence number in the message to discard any duplicates. This helps ensure that no messages are lost, and provides fault-tolerance for publishers.

The AMPS C++, Java, C# and Python clients each provide different implementation of persistent subscriptions and persistent publication. Please refer to the *High Availability* chapter of the *Client Development Guide* for the language of your choice to see how this feature is implemented.

Heartbeat in High Availability

Use of the heartbeat feature 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 server and the AMPS clients 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 server assumes disconnection has occurred if these regular heartbeats cease, ensuring disconnection is detected in a timely manner.

Heartbeats are initialized by the AMPS client by sending a `heartbeat` message to the AMPS server. To enable heartbeats in your application, refer to the *High Availability* chapter in the Developer Guide for your specific client language.

Slow Client Management

Sometimes, AMPS can publish messages faster than an individual client can consume messages, particularly in applications where the pattern of messages includes "bursts" of mesages. Clients that are unable to consume messages faster or equal to the rate messages are being sent to them are ”slow clients”. By default, AMPS queues messages for a slow client in memory to grant the slow client the opportunity to catch up. However, scenarios may arise where a client can be “over-subscribed” to the point that the client cannot consume messages as fast as messages are being sent to it. In particular, this can happen with the results of a large SOW query, where AMPS generates all of the messages for the query much faster than the network can transmit the messages.

Slow client management is one of the ways that AMPS prevents slow clients from disrupting service to the instance. 60East recommends enabling slow client management for instances that serve high message volume or are mission critical.

There are two methods that AMPS uses for managing slow clients to minimize the effect of slow clients on the AMPS instance:

- *Client offlining*. When client offlining occurs, AMPS buffers the messages for that client to disk. This relieves pressure on memory, while allowing the client to continue processing messages.
- *Disconnection*. When disconnection occurs, AMPS closes the client connection, which immediately ends any subscriptions, in-progress `sow` queries, or other commands from that client. AMPS also removes any offlined messages for that client.

AMPS provides resource pool protection, to protect the capacity of the instance as a whole, and client-level protection, to identify unresponsive clients.

Resource Pool Policies

AMPS uses resource pools for memory and disk consumption for clients. When the memory limit is exceeded, AMPS chooses a client to be offlined. When the disk limit is exceeded, AMPS chooses a client to be disconnected.

When choosing which client will be offlined or disconnected, AMPS identifies the client that uses the largest amount of resources (memory and/or disk). That client will be offlined or disconnected.

AMPS allows you to use a global resource pool for the entire instance, a resource pool for each transport, or any combination of the two approaches. By default, AMPS configures a global resource pool that is shared across all transports. When an individual transport specifies a different setting for a resource pool, that transport receives an individual resource pool. For example, you might set high resource limits for a particular transport that serves a mission-critical application, allowing connections from that application to consume more resources than connections for less important applications.

The following table shows resource pool options for slow client management:

Table 25.3. Slow Client: Resource Pool Policies

Element	Description
`MessageMemoryLimit`	The total amount of memory to allocate to messages before offlining clients. Default: 10% of total host memory or 10% of the amount of host memory AMPS is allowed to consume (as reported by `ulimit -m`), whichever is *lowest*.
`MessageDiskLimit`	The total amount of disk space to allocate to messages before disconnecting clients.

Element	Description
	Default: 1GB or the amount specified in the MessageMemoryLimit, whichever is *highest*.
`MessageDiskPath`	The path to use to write offline files. Default: `/tmp`

Individual Client Policies

AMPS also allows you to set policies that apply to individual clients. These policies are applied to clients independently of the instance level policies. For example, a client that exceeds the capacity limit for an individual client will be disconnected, even if the instance overall has enough capacity to hold messages for the client.

As with the Resource Pool Policies, Transports can either use instance-level settings or create settings specific to that transport.

The following table shows the client level options for slow client management:

Table 25.4. Slow Client: Individual Client Policies

Element	Description
`ClientMessageAgeLimit`	The maximum amount of time for the client to lag behind. If a message for the client has been held longer than this time, the client will be disconnected. This parameter is an AMPS time interval (for example, `30s` for 30 seconds, or `1h` for 1 hour). Default: No age limit
`ClientMaxCapacity`	The amount of available capacity a single client can consume. Before a client is offlined, this limit applies to the `MessageMemoryLimit`. After a client is offlined, this limit applies to the `MessageDiskLimit`. This parameter is a percentage of the total. Default: `100%` (no effective limit)

Client offlining can require careful configuration, particularly in situations where applications retrieve large result sets from SOW queries when the application starts up. More information on tuning slow client offlining for AMPS is available in the section called “Slow Client Offlining for Large Result Sets”.

Configuration Example

```
<AMPSConfig>

 ...


 <MessageMemoryLimit>10GB</MessageMemoryLimit>
 <MessageDiskPath>/mnt/fastio/AMPS/offline</MessageDiskPath>
 <ClientMessageAgeLimit>30s</ClientMessageAgeLimit>
```

```
...
  <Transports>

  <!-- This transport shares the 10GB MessageMemoryLimit
       defined for the instance. -->
  <Transport>
    <Name>regular-json-tcp</Name>
    <Type>tcp</Type>
    <InetAddr>9007</InetAddr>
    <ReuseAddr>true</ReuseAddr>
    <MessageType>json</MessageType>
  </Transport>

  <!-- This transport shares the 10GB MessageMemoryLimit
       defined for the instance. -->
  <Transport>
    <Name>regular-bson-tcp</Name>
    <Type>tcp</Type>
    <InetAddr>9010</InetAddr>
    <ReuseAddr>true</ReuseAddr>
    <MessageType>bson</MessageType>
    <!-- However, this transport does not allow
         clients to fall as far behind as the
         instance-level setting -->
    <ClientMessageAgeLimit>15s</ClientMessageAgeLimit>
  </Transport>


  <!-- This transport has a separate 35GB MessageMemoryLimit
       and a 70GB MessageDiskLimit. It uses the instance-wide
       30s parameter for the ClientMessageAgeLimit -->
  <Transport>
    <Name>highpri-json-tcp</Name>
    <Type>tcp</Type>
    <InetAddr>9995</InetAddr>
    <ReuseAddr>true</ReuseAddr>
    <MessageType>json</MessageType>
    <MessageMemoryLimit>35GB</MessageMemoryLimit>
    <MessageDiskLimit>70GB</MessageDiskLimit>
  </Transport>

  </Transports>

</AMPSConfig>
```

Example 25.3. Transports Example with Resource Management

Message Ordering and Replication

AMPS uses the name of the publisher and the sequence number assigned by the publisher to ensure that messages from each publisher are published in order. However, AMPS does not enforce order across publishers. This means that, in a failover situation, that messages from different publishers may be interleaved in a different order on different servers, even though the message stream from each publisher is preserved in order. Each instance preserves the order in which messages were processed by that instance, and enforces that order.

25.5. Replicated Queues

AMPS provides a unique approach to replicating queues. This approach is designed to offer high performance in the most common cases, while continuing to provide delivery model guarantees, resilience and failover in the event that one of the replicated instances goes offline.

When a queue is replicated, AMPS replicates the `publish` commands to the underlying topic, the `sow_delete` commands that contain the acknowledgement messages, and special queue management commands that are internal to AMPS.

Queue Message Ownership

To guarantee that no message is delivered more than once, AMPS tracks ownership of the message within the network of replicated instances. When a message is first published to AMPS, the instance that recieves the publish command owns the message. Although all replicated instances downstream instances record the publish command in their transaction logs, they do not provide the message to queue subscribers unless that instance owns the message.

Only one instance can own a message at any given time. To transfer ownership, an instance that does not currently own the message makes a request to the current message owner. The owning instance makes an explicit decision to transfer ownership, and replicates the transfer notification to all instances to which the queue topic is replicated.

The instance that owns a message will always deliver the message to a local subscriber if possible. This means that performance for local subscribers is unaffected by the number of downstream instances. However, this also means that if the local subscribers are keeping up with the message volume being published to the queue, the owning instance will never need to grant a transfer of ownership.

Downstream instances can request that the owner transfer ownership of a message.

A downstream instance will make this request if:

1. The downstream instance has subscriptions for that topic with available backlog, *and*
2. The amount of time since the message arrived at the instance is greater than the typical time between the replicated message arriving and the replicated acknowledgement arriving.

Notice that this approach is intended to minimize ungranted transfer requests. In normal circumstances, the typical processing time reflects the speed at which the local processors are consuming messages at a steady state. Downstream instances will only request messages that have been seen to exceed that time, indicating that the processors are not keeping up with the incoming message rate.

The instance that owns the message will grant ownership to a requesting instance if:

1. The request is the first request received for this message, *and*
2. There are no subscribers on the owning instance that can accept the message

When the owning instance grants the request, it logs the transfer in its transaction log and sends the transfer of ownership to all instances that are receiving replicated messages for the queue. When the owning instance does not grant the transfer of ownership, it takes no action.

Notice that your replication topology must be able to replicate acknowledgements to all instances that receive messages for the queue. Otherwise, an instance that does not receive the acknowledgements will not consider the messages to be processed. Replication validation can help to identify topologies that do not meet this requirement.

Failover and Queue Message Ownership

When an instance that contains a queue fails or is shut down, that instance is no longer able to grant ownership requests for the messages that it owns. By default, those messages become unavailable for delivery, since there is no longer a central coordination point at which to ensure that the messages are only delivered once.

AMPS provides a way to make those messages available. Through the admin console, you can choose to `enable_proxied_transfer`, which allows an instance to act as an ownership proxy for an instance that has gone offline. In this mode, the local instance can assume ownership of messages that is owned by an offline instance.

Use this setting with care: when active, it is possible for messages to be delivered twice if the instance that had previously owned the message comes back online, or if multiple instances have proxied transfer enabled for the same queue.

In general, you `enable_proxied_transfer` as a temporary recovery step while one of the instances is offline, and then disable proxied transfer when the instance comes back online, or when all of the messages owned by that instance have been processed.

Configuration for Queue Replication

To replicate queues, AMPS requires that the replication configuration:

1. Provide bidirectional replication between the instances. In other words, if instance A replicates a queue to instance B, instance B must also replicate that queue to instance A.

2. If the topic is a queue on one instance, it must be a queue on all replicated instances.

3. On all replicated instances, the queue must use the same underlying topic definition and filters. For queues that use a regular expression as the topic definition, this means that the regular expression must be the same.

4. Replicated instances must provide passthrough for instances that replicate queues. For example, consider the following replication topology: Instance A in group One replicates a queue to instance B in group Two. Instance B in group Two replicates the queue to instance C in group Three.

   For this configuration, instance B must provide passthrough for group Three to instance A, and must also provide passthrough for group One to instance C. The reason for this is to ensure that ownership transfer and acknowledgement messages can reach all instances that maintain a copy of the queue.

Notice that the requirements above apply only to queue topics. If the underlying topic uses a different name than the queue topic, it is possible to replicate the messages from the underlying topic without replicating the queue itself. This approach can be convenient for simply recording and storing the messages provided to the queue on an archival or auditing instance. When only the underlying topic (or topics) are replicated, the requirements above do not apply, since AMPS does not provide queuing behavior for the underlying topics.

Chapter 26. Operation and Deployment

This chapter contains guidelines and best-practices to help plan and prepare an environment to meet the demands that AMPS is expected to manage.

26.1. Capacity Planning

Sizing an AMPS deployment can be a complicated process that includes many factors including configuration parameters used for AMPS, the data used within the deployment, and how the deployment will be used. This section presents guidelines that you can use in sizing your host environment for an AMPS deployment given what needs to be considered along every dimension: Memory, Storage, CPU, and Network.

Memory

Beyond storing its own binary images in system memory, AMPS also tries to store its SOW and indexing state in memory to maximize the performance of record updates and SOW queries.

AMPS needs less than 1GB for its own binary image and initial start up state for most configurations. In the worst-case, because of indexing for queries, AMPS may need up to twice the size of messages stored in the SOW. And, finally AMPS needs some amount of memory reserved for the clients connected to it. While the per connection overhead is a tunable parameter based on the Slow Client Disconnect settings (see the best practices later in this chapter) it is advised to use 50MB per connected client.

This puts the worst-case memory consumption estimate at:

Equation 26.1. Memory estimation equation

$$1GB + (2S * M) + (C * 50MB)$$

where:

$$S = \text{Average SOW Message Size}$$

$$M = \text{Number of SOW Messages}$$

$$C = \text{Number of Clients}$$

Equation 26.2. Example memory estimation equation

$$1GB + (2 * 1024 * 20{,}000{,}000) + (200 * 50MB) \approx 52GB$$

where:

$$S = 1024$$

$$M = 20{,}000{,}000$$

$$C = 200$$

An AMPS deployment expected to hold 20 million messages with an average message size of 1KB and 200 connected clients would consume 52GB. Therefore, this AMPS deployment would fit within a host containing 64GB with enough headroom for the OS under most configurations.

Storage

AMPS needs enough space to store its own binary images, configuration files, SOW persistence files, log files, transaction log journals, and slow client offline storage, if any. Not every deployment configures a SOW or transaction log, so the storage requirements are largely driven by the configuration.

AMPS log files. Log file sizes vary depending on the log level and how the engine is used. For example, in the worst-case, `trace` logging, AMPS will need at least enough storage for every message published into AMPS and every message sent out of AMPS plus 20%.

For `info` level logging, a good estimate of AMPS log file sizes would be 2MB per 10 million messages published.

Logging space overhead can be capped by implementing a log rotation strategy which uses the same file name for each rotation. This strategy effectively truncates the file when it reaches the log rotation threshold to prevent it from growing larger.

SOW . When calculating the SOW storage, there are a couple of factors to keep in mind. The first is the average size of messages stored in the SOW, the number of messages stored in the SOW and the `SlabSize` defined in the configuration file. Using these values, it is possible to estimate the minimum and maximum storage requirements for the SOW:

Equation 26.3. Minimum SOW Size

$$\text{Min} = (\text{MsgSize} * \text{MsgCount}) + (\text{Cores} * \text{SlabSize})$$

where

$$\text{Min} = \text{Minimum SOW Size}$$

$$\text{MsgSize} = \text{Average SOW Message Size}$$

$$\text{MsgCount} = \text{Number of SOW Messages}$$

$$\text{SlabSize} = \text{Slab size for the SOW}$$

$$\text{Cores} = \text{Number of processor cores in the system}$$

Equation 26.4. Maximum SOW Size

$$\text{Max} = \left(\text{MsgSize} + \text{SlabSize}\right) * \text{MsgCount}$$

where

$$\text{Max} = \text{Maximum SOW Size}$$

$$\text{MsgSize} = \text{Average SOW Message Size}$$

$$\text{SlabSize} = \text{Slab size for the SOW}$$

$$\text{MsgCount} = \text{Number of SOW Messages}$$

The storage requirements should be between the two values above, however it is still possible for the SOW to consume additional storage based on the unused capacity configured for each SOW topic. Further, notice that AMPS reserves the configured `SlabSize` for each processor core in the system the first time a thread running on that core writes to the SOW.

For example, in an AMPS configuration file with the `SlabSize` set to 1MB, the SOW for this topic will consume 1MB per processor core with no messages stored in the SOW. Pre-allocating SOW capacity in chunks, as a chunk is needed, is more efficient for the operating system, storage devices, and helps amortize the SOW extension costs over more messages.

It is also important to be aware of the maximum message size that AMPS guarantees the SOW can hold. The maximum message size is calculated in the following manner:

Equation 26.5. Maximum Message Size allowed in SOW

$$Max = SlabSize - 64 bytes$$

where

$$Max = \text{Maximum SOW Size}$$

$$SlabSize = \text{The configured SlabSize for the SOW}$$

This calculation says that the maximum message size that can be stored in the SOW in a single message storage is the `SlabSize` minus 64 bytes for the record header information. AMPS enforces a lower limit of approximately 1MB: if the maximum size works out to less than 1MB, AMPS will use 1MB as the maximum size for the topic.

Transaction logs. Transaction logs are used for message replay, replication, and to ensure consistency in environments where each message is critical. Transaction logs are optional in AMPS, and transaction logs can be configured for individual topics or filters. When planning for transaction logs, there are three main considerations:

- The total size needed for the transaction log
- The size to allow for each file that makes up the transaction log
- How many files to preallocate

You can calculate the approximate total size of the transaction log as follows:

Equation 26.6. Transaction Log Sizing Approximation

$$Capacity = (S + 512 bytes) * N$$

where

$$Capacity = \text{Estimated storage capacity required for transaction log}$$

$$S = \text{Average message size}$$

$$N = \text{Number of messages to retain}$$

Size your files to match the aging policy for the transaction log data. To remove data from the transaction log, you simply archive or delete files that are no longer needed. You can size your files to make this easier. For example, if your application typically generates 100GB a day of transaction log, you could size your files in 10GB units to make it easier to remove 100GB increments.

AMPS allows you to preallocate files for the transaction log. For applications that are very latency-sensitive, preallocation can help provide consistent latency. We recommend that those applications preallocate files, if storage capacity and retention policy permit. For example, an application that sees heavy throughput during a working day might preallocate enough files so that there is no need for additional allocation within the working day.

Other Storage Considerations. The previous sections discuss the scope of sizing the storage, however scenarios exist where the performance of the storage devices must also be taken into consideration.

One such scenario is the following use case in which the AMPS transaction log is expected to be heavily used. If performance greater than 50MB/second is required out of the AMPS transaction log, experience has demonstrated that flash storage (or better) would be recommended. Magnetic hard disks lack the performance to produce results greater than this with a consistent latency profile.

CPU

SOW queries with content filtering make heavy use of CPU-based operations and, as such, CPU performance directly impacts the content filtering performance and rates at which AMPS processes messages. The number of cores within a CPU largely determines how quickly SOW queries execute.

AMPS contains optimizations which are only enabled on recent 64-bit x86 CPUs. To achieve the highest level performance, consider deploying on a CPU which includes support for the SSE 4.2 instruction set.

To give an idea of AMPS performance, repeated testing has demonstrated that a moderate query filter with 5 predicates can be executed against 1KB messages at more than 1,000,000 messages per second, per core on an Intel i7 3GHz CPU. This applies to both subscription based content filtering and SOW queries. Actual messaging rates will vary based on matching ratios and network utilization.

Network

When capacity planning a network for AMPS, the requirements are largely dependent on the following factors:

- average message size
- the rate at which publishers will publish messages to AMPS
- the number of publishers and the number of subscribers.

AMPS requires sufficient network capacity to service inbound publishing as well as outbound messaging requirements. In most deployments, outbound messaging to subscribers and query clients has the highest bandwidth requirements due to the increased likeliness for a “one to many” relationship of a single published message matching subscriptions/queries for many clients.

Estimating network capacity requires knowledge about several factors, including but not limited to: the average message size published to the AMPS instance, the number of messages published per second, the average expected match ratio per subscription, the number of subscriptions, and the background query load. Once these key metrics are known, then the necessary network capacity can be calculated:

Equation 26.7. Network capacity formula

$$\mathrm{R} * \mathrm{S}(1 + \mathrm{M} * \mathrm{S}) + \mathrm{Q}$$

where

$$\mathrm{R} = \mathrm{Rate}$$

$$\mathrm{S} = \mathrm{Average\ Message\ Size}$$

$$\mathrm{M} = \mathrm{Match\ Ratio}$$

$$\mathrm{Q} = \mathrm{Query\ Load}$$

where “Query Load” is defined as:

$$M_q * \mathrm{S} * Q_s$$

where

$$M_q = \mathrm{Messages\ Per\ Query}$$

$S = \text{Average Message Size}$

$Q_s = \text{Queries Per Second}$

In a deployment required to process published messages at a rate of 5000 messages per second, with each message having an average message size of 600 bytes, the expected match rate per subscription is 2% (or 0.02) with 100 subscriptions. The deployment is also expected to process 5 queries per 1 minute (or 12 queries per second), with each query expected to return 1000 messages.

$$5000 * 600\mathrm{B} * \left(1 + 0.02 * 100\right) + \left(1000 * 600\mathrm{B} * \frac{1}{12}\right) \approx 9\mathrm{MB/s} \approx 72\mathrm{Mb/s}$$

Based on these requirements, this deployment would need at least 72Mb/s of network capacity to achieve the desired goals. This analysis demonstrates AMPS by it self would fall into a 100Mb/s class network. It is important to note, this analysis does not examine any other network based activity which may exist on the host, and as such a larger capacity networking infrastructure than 100Mb/s would likely be required.

NUMA Considerations

AMPS is designed to take advantage of non-uniform memory access (NUMA). For the lowest latency in networking, we recommend that you install your NIC in the slot closest to NUMA node 0. AMPS runs critical threads on node 0, so positioning the NIC closest to that node provides the shortest path from processor to NIC.

26.2. Linux Operating System Configuration

This section covers some settings which are specific to running AMPS on a Linux Operating System.

ulimit

The `ulimit` command is used by a Linux administrator to get and set user limits on various system resources.

ulimit -c. It is common for an AMPS instance to be configured to consume gigabytes of memory for large SOW caches. If a failure were to occur in a large deployment it could take seconds (maybe even hours, depending on storage performance and process size!) to dump the core file. AMPS has a minidump reporting mechanism built in that collects information important to debugging an instance before exiting. This minidump is much faster than dumping a core file to disk. For this reason, it is recommended that the per user core file size limit is set to 0 to prevent a large process image from being dumped to storage.

ulimit -n. The number of file descriptors allowed for a user running AMPS needs to be at least double the sum of counts for the following: connected clients, SOW topics and pre-allocated journal files. *Minimum: 4096. Recommended: 32768, or the value recommended by AMPS in any diagnostic messages, whichever is greater*

/proc/sys/fs/aio-max-nr

Each AMPS instance requires AIO in the kernel to support at least 16384 plus 8192 for each SOW topic in simultaneous I/O operations. The setting `aio-max-nr` is global to the host and impacts all applications. As such this value needs to be set high enough to service all applications using AIO on the host. *Minimum: 65536. Recommended: 1048576*

To view the value of this setting, as root you can enter the following command:

```
cat /proc/sys/fs/aio-max-nr
```

To edit this value, as root you can enter the following command:

```
sysctl -w fs.aio-max-nr = 1048576
```

This command will update the value for `/proc/sys/fs/aio-max-nr` and allow 1,048,576 simultaneous I/O operations, but will only do so until the next time the machine is rebooted. To make a permanent change to this setting, as a root user, edit the `/etc/sysctl.conf` file and either edit or append the following setting:

```
fs.aio-max-nr = 1048576
```

/proc/sys/fs/file-max

Each AMPS instance needs file descriptors to service connections and maintain file handles for open files. This number needs to be at least double the sum of counts for the following: connected clients, SOW topics and pre-allocated journal files. This file-max setting is global to the host and impacts all applications, so this needs to be set high enough to service all applications on the host. *Minimum: 262144 Recommended: 6815744*

To view the value of this setting, as root you can enter the following command:

```
cat /proc/sys/fs/file-max
```

To edit this value, as root you can enter the following command:

```
sysctl -w fs.file-max = 6815744
```

This command will update the value for `/proc/sys/fs/file-max` and allow 6,815,744 concurrent files to be opened, but will only do so until the next time the machine is rebooted. To make a permanent change to this setting, as a root user, edit the `/etc/sysctl.conf` file and either edit or append the following setting:

```
fs.file-max = 6815744
```

26.3. Upgrading an AMPS Installation

This chapter describes how to upgrade an existing installation of AMPS. The steps presented here focus on upgrading the installation itself, and should be the only steps you need for upgrades that change the HOTFIX version number or the MAINTENANCE version number (as described in Table 1.2).

For changes that update the MAJOR or MINOR version number, AMPS may add features, change file or network formats, or change behavior. For these upgrades, you may need to make changes to the AMPS configuration file or update applications to adapt to new features or changes in behavior.

60East recommends maintaining a test environment that you can use to test upgrades, particularly when an upgrade changes MAJOR or MINOR versions and you are taking advantage of new features or changed behavior.

When the AMPS instance participates in replication, you must coordinate the instance upgrades when upgrading across AMPS versions. AMPS replication works between instances with the same major and minor version number (for example, all AMPS 3.9 releases use the same version of replication, but the 4.0 releases use a different version of replication.) When the AMPS instance participates in replication, you must coordinate the instance upgrades when upgrading across AMPS versions. AMPS replication works between instances with the same major and minor version number (for example, all AMPS 3.9 releases use the same version of replication, but the 4.0 releases use a different version of replication.)

Upgrade Steps

Upgrading an AMPS installation involves the following steps:

1. Stop the running instance
2. If necessary, upgrade any data files or configuration files that you want to retain
3. If necessary, update any applications that will use new features
4. Install the new AMPS binaries
5. Restart the service

As mentioned above, if you are using replication, and the upgrade increments the MAJOR or MINOR version number, you must upgrade all of the instances that replicate at the same time for replication to suceed. This is typically accomplished with a rolling upgrade, where instances are upgraded on a specific schedule to minimize downtime.

Upgrading AMPS Data Files

AMPS may change the format and content of data files when upgrading across versions, as specified by the major and minor version number. This most commonly occurs when new features are added to AMPS that require different or additional information in the persisted files. The HISTORY file for the AMPS release lists when changes have been made that require data file changes.

In general, 60East recommends upgrading the data files whenever moving to a new major/minor version and whenever a data file change is mentioned in the HISTORY file.

The AMPS distribution includes the `amps_upgrade` utility to process and upgrade data files. The version included with each release of AMPS upgrades previous versions of the data files to the version of AMPS that includes the utility. For example, the version of `amps_upgrade` included in version 4.1 of AMPS upgrades files to the 4.1 version the data files.

AMPS versions may upgrade any of the following types of files:

- *journals* - these files contain the transaction logs for the instance
- *clients.ack* - this file contains a cache of the last sequence number processed for a publisher

- *sow files* - these files contain the persisted state of the durable SOW topics for the instance

The `amps_upgrade` utility handles upgrades for each of these types of files. Full details on `amps_upgrade` are available in the *AMPS Utilities Guide*.

26.4. Best Practices

This section covers a selection of best practices for deploying AMPS.

Monitoring

AMPS exposes the statistics available for monitoring via a RESTful interface, known as the Monitoring Interface, which is configured as the administration port. This interface allows developers and administrators to easily inspect various aspects of AMPS performance and resource consumption using standard monitoring tools.

At times AMPS will emit log messages notifying that a thread has encountered a deadlock or stressful operation. These messages will repeat with the word “stuck” in them. AMPS will attempt to resolve these issues, however after 60 seconds of a single thread being stuck, AMPS will automatically emit a minidump to the previously configured minidump directory. This minidump can be used by 60East support to assist in troubleshooting the location of the stuck thread or the stressful process.

Another area to examine when monitoring AMPS is the `last_active` monitor for the processors. This can be found in the `/amps/instance/processors/all/last_active` url in the monitoring interface. If the `last_active` value continually increases for more than one minute and there is a noticeable decline in the quality of service, then it may be best to fail-over and restart the AMPS instance.

Stopping AMPS

To stop AMPS, ensure that AMPS runs the `amps-action-do-shutdown` action. By default, this action is run when AMPS receives `SIGHUP`, `SIGINT`, or `SIGTERM`. However, you can also configure an Action to shut down AMPS in response to other conditions. For example, if your company policy is to reboot servers every Saturday night, and AMPS is not running as a system service (or daemon), you could schedule an AMPS shutdown every Saturday before the system reboot.

When AMPS is installed to run as a system service (or daemon), AMPS installs shutdown scripts that will cleanly stop AMPS during a system shutdown or reboot.

SOW Parameters

Choosing the ideal `SlabSize` for your SOW topic is a balance between the frequency of SOW expansion and storage space efficiency. A large `SlabSize` will preallocate space for records when AMPS begins writing to the SOW.

If detailed tuning is not necessary, 60East recommends leaving the `SlabSize` at the default size if your messages are smaller than the default `SlabSize`. If your messages are larger than the default SlabSize, a good starting point for the `SlabSize` is to set it to several times the maximum message size you expect to store in the SOW.

There are three considerations when setting the optimium `SlabSize`:

- Frequency of allocations
- Overall size of the SOW
- Efficient use of space

A `SlabSize` that is small results in frequent extensions of your SOW topic to occur. These frequent extensions can reduce throughput in a heavily loaded system, and in extreme cases can exhaust the kernel limit on the number of regions that a process can map. Increasing the `SlabSize` will reduce the number of allocations.

When the `SlabSize` is large, then the risk of the SOW resize affecting performance is reduced. Since each slab is larger, however, there will be more space consumed if you are only storing a small number of messages: this cost will amortize as the number of messages in the SOW exceeds the *number of cores in the system * the number of messages that fit into a slab*.

To most efficiently use space, set a `SlabSize` that minimizes the amount of unused space in a slab. For example, if your message sizes are average 512 bytes but can reach a maximum of 1.2 MB, one approach would be to set a `SlabSize` of 2.5MB to hold approximately 5 average-sized messages and two of the larger-sized messages. Looking at the actual distribution of message sizes in the SOW (which can be done with the `amps_sow_dump` utility) can help you determine how best to size slabs for maximum space efficiency.

For optimizing the `SlabSize`, determine how important each aspect of SOW tuning is for your application, and adjust the configuration to balance allocation frequency, overall SOW size, and space to meet the needs of your application.

Slow Clients

As described in the section called “Slow Client Management ”, AMPS provides capacity limits for slow clients to reduce the memory resources consumed by slow clients. This section discusses tuning slow client handling to achieve your availability goals.

Slow Client Offlining for Large Result Sets

The default settings for AMPS work well in a wide variety of applications with minimal tuning.

If you have particularly large SOW topics, and your application is disconnecting clients due to exceeding the offlining threshold when the clients retrieving large SOW query result sets, 60East recommends the following settings as a baseline for further tuning:

Table 26.1. Client Offline Settings for Large Result Sets

Parameter	Recommendation
`MessageMemoryLimit`	This controls the maximum memory consumed by AMPS for client messages. You can increase this parameter to allow AMPS to use more memory to records. Notice, however, that memory devoted to client messages is unavailable for other purposes. Recommended starting point for tuning large result sets: 10%. 60East recommends tuning the `Mes-`

Parameter	Recommendation
	sageDiskLimit first. If necessary, increase this parameter by 1-2% at a time. Use caution with settings over 20%: devoting large amounts of memory to client messages may cause swapping and reduce, rather than increase, overall performance.
MessageDiskLimit	The maximum amount of space to consume for offline messages. Recommended starting point for tuning large result sets: Average record size * number of expected records * number of simultaneous clients, or MessageMemoryLimit, whichever is greater.
MessageDiskPath	The path in which to store offline message files. 60East recommends that the message disk path be hosted on fast, high-capacity storage such as a PCIe-attached flash drive. The available storage capacity of the disk must be greater than the configured MessageDiskLimit. Pay attention to the performance characteristics of the device: for example, some devices suffer reduced performance when they run low on free space, so for those devices you would want to make sure that there is space available on the device even when AMPS is close to the MessageDiskLimit.

60East recommends that you use these settings as a baseline for further tuning, bearing in mind the needs and expected messaging patterns of your application.

Minidump

AMPS includes the ability to generate a minidump file which can be used in support scenarios to attempt to troubleshoot a problematic instance. The minidump captures information prior to AMPS exiting and can be obtained much faster than a standard core dump (see the section called “ulimit” for more configuration options). By default the minidump is configured to write to `/tmp`, but this can be changed in the AMPS configuration by modifying the `MiniDumpDirectory`.

Minidumps contain thread state information that provides the location of each running thread and register information for the thread. The minidump also contains basic information about the system that AMPS was running on, such as the processor type and number of sockets. Minidumps do not contain the full internal state of AMPS or the full contents of application memory. Instead, minidumps identify the point of failure to help 60East quickly narrow down the issue without generating large files or potentially compromising sensitive data.

Generation of a minidump file occurs in the following ways:

1. When AMPS detects a crash internally, a minidump file will automatically be generated.
2. When a user clicks on the `minidump` link in the `amps/instance/administrator` link from the administrator console (see the *AMPS Monitoring Reference* for more information).
3. By sending the running AMPS process the `SIGQUIT` signal.

4. If AMPS observes a single stuck thread for 60 seconds, a minidump will automatically be generated. This should be sent to AMPS support for evaluation along with a description of the operations taking place at the time.

Chapter 27. Securing AMPS

One of the most important considerations when using AMPS in production is keeping your data safe. This means both ensuring that subscribers only have access to the data that they are allowed to have and that only authorized publishers are allowed to publish messages into the system. This chapter describes the mechanisms within AMPS to protect access to AMPS resources through client, administrative, and replication connections.

In this chapter, we describe the AMPS security infrastructure and present general information about securing an AMPS installation. AMPS uses a plugin model for providing authentication and entitlement, and allows a great deal of freedom in how the a given module implements security checks. This chapter discusses the concepts, principles, and guarantees that AMPS provides. The specific steps and configuration you use to secure an installation of AMPS depend on the plugin you use to secure AMPS.

There are three aspects to securing connections to AMPS:

- Authentication assigns an identity to a connection and verifies that identity
- Entitlement enforces permission to access AMPS and read or write AMPS resources based on the identity assigned to a connection
- The AMPS process may also need to provide credentials to another AMPS instance (for example, to secure outgoing replication)

AMPS installations typically create custom plugins for securing AMPS. These plugins integrate with the enterprise authentication and entitlement system, and are designed to enforce the policies for the specific site. For more information on developing modules for use with AMPS, contact 60East support for the AMPS Server SDK.

27.1. Authentication

The first part of securing AMPS is developing a strategy to verify the identity of connected clients. AMPS maintains an identity for each client connection, and uses that identity for entitlement requests. Once an identity is assigned to a connection, that identity stays the same for the lifetime of the connection. If an application needs to use different identities to work with AMPS, that application needs to make a separate connection for each identity.

There are two ways that AMPS assigns an identity to a client:

1. When an application explicitly sends a `logon` command, AMPS uses the credentials in the message for the authentication process. If authentication is successful, AMPS associates the user name provided in the initial logon with the connection. If authentication fails, AMPS closes the connection.

2. When an application issues any other command after connecting but before sending a `logon` command, AMPS treats this as an *implicit logon* and begins the authentication process with an empty user name and password. If authentication is successful, AMPS associates an empty user name with the connection. If authentication fails, AMPS closes the connection. AMPS does not allow implicit logon by default in 5.0 and later versions. However, you can enable implicit logon as described below.

In both cases, authentication occurs through the AMPS security infrastructure.

When authenticating a client, AMPS locates the authentication module in use for client's transport (or, for the admin interface, the special `amps-admin` transport). If there is an authentication module specified for that `Transport`, AMPS uses that module. Otherwise, the transport uses an instance of the authentication module specified for the instance. When the configuration for the instance doesn't include an instance level authentication module, the default module for the transport is `amps-default-authentication-module`, which requires a logon, but accepts any user name and password provided and sets the authenticated user name to an empty string.

Once AMPS has located the module instance, AMPS provides the user name and the password to that instance of the module. The module can accept the credentials, reject the credentials, or return a challenge that the application must respond to. When the module returns a challenge, the connection remains unauthenticated until the application requesting authentication responds to the challenge and the module accepts the response.

For most production systems, AMPS security is integrated with the overall security fabric of the organization. 60East provides the *AMPS Server SDK* to help developers create authentication modules that implement the unique policies and procedures required by a particular organization.

Simple Authentication Modules

AMPS includes three simple authentication modules in the AMPS distribution. These modules provide very simple policies for authentication, and are most useful in testing and development environments.

Table 27.1. Simple Authentication Modules

Module	Description
`amps-default-authentication-module`	Allows any user name and password. Does not allow implicit logon by default. Does not provide the user name to AMPS by default.
`amps-implicit-authentication-module`	Allows any user name and password. Allows implicit logon by default. Does not provide the user name to AMPS by default.
`amps-default-no-authentication-module`	Does not allow authentication regardless of the username and password provided. This can be useful for testing application behavior when logon is denied, or for setting a policy for the instance that individual transports must override.

Enabling Implicit Logon

60East recommends using explicit logon commands in your applications wherever possible, and the default authentication module disallows implicit logons. For backward compatibility with older versions of AMPS, AMPS includes the `amps-implicit-authentication-module` which allows implicit logon to restore the behavior of the previous AMPS versions. To use the `amps-implicit-authentication-module` for all of the transports in the instance, set the instance-level Authentication to use this module, as shown below:

```
<AMPSConfig>
  ...
  <Authentication>
     <Module>amps-implicit-authentication-module</Module>
  </Authentication>
  ...
</AMPSConfig>
```

27.2. Entitlement

The AMPS entitlement system controls access to individual resources in AMPS. Each entitlement request consists of a user, a specific action, and, where applicable, the type of resource and the resource name. For example, an entitlement request might arrive for the user `Janice` to `write` (that is, publish) to the `topic` named `/orders/northamerica`. Another entitlement request might be for the user `Phil` to `logon` to the instance. A third request might be for the user `Jill` to `read` (that is, subscribe or run a SOW query) from the topic named `/orders/pacific/palau`.

When checking entitlements, AMPS locates the entitlement module in use for the Transport that the client is connecting on (or, for the Admin interface, the special `amps-admin` transport). If there is an entitlement module specified for the Transport, AMPS uses that module. Otherwise, AMPS uses an instance of the entitlement module specified for the instance. When the configuration file for the instance doesn't specify an instance-level entitlement module, the default module for the transport is `amps-default-entitlement-module`, which allows all permissions for any user.

AMPS caches the results of the entitlement check. You can clear the entitlement cache for all users using the AMPS Administrative Actions. You can clear the entitlement cache for a single user using the AMPS external API. When the entitlement cache is cleared, AMPS disconnects the user. This ensures that, when the user reconnects, the user only has access to resources that match the current set of entitlements.

AMPS checks entitlements for a command when processing the command, and does not recheck permissions after the command is processed. For example, when `Jill` subscribes to `/orders/pacific/palau`, AMPS checks entitlements when creating the subscription. If the entitlement check returns an entitlement content filter, AMPS includes that entitlement filter on the subscription. Once the subscription has been created, AMPS applies the filter as a part of the standard filtering process, but AMPS does not check entitlements for the subscription as further messages arrive.

The following table lists the resource types that AMPS provides:

Table 27.2. AMPS Entitlement Resource Types

Resource Type	Description
`logon`	Permission to log on to the AMPS instance
`replication_logon`	Permission to log on to the AMPS instance as a replication source
`topic`	Permission to receive from or publish to a specific topic
`admin`	Permission to read admin statistics or peform admin functions from the web interface

For the `topic` and `admin` resource types, AMPS also provides the name of the resource and whether the request is to `read` the resource or `write` to the resource.

The table below shows how AMPS commands translate to entitlement types:

Table 27.3. Entitlement Types for Commands

AMPS Command	Entitlement Type
`delta_subscribe`, `sow`, `sow_and_subscribe`,	`read`

AMPS Command	Entitlement Type
`subscribe`, `sow_and_delta_subscribe`	
`delta_publish`, `publish`, `sow_delete`	`write`
commands received over replication	`replication allowed`

Entitlement Caching

AMPS does not present a request to the entitlement module each time that an entitlement check is needed. Instead, AMPS presents the request the first time the entitlement is needed, and then caches the results from the module for subsequent entitlement checks. This improves performance, although it also means that when a module that reads entitlements from an external source (such as a central directory of permissions) that may change without requiring a restart of the AMPS instance, that module will need to establish a policy for resetting the entitlement cache.

Regular Expression Subscriptions

Each request from AMPS is for a specific resource name. When a client requests a regular expression subscription, AMPS makes a request for each topic that matches the subscription at the point that AMPS has a message to deliver for that topic. For example, if the user `Nina` enters a subscription for `/parts/(mechanical|electrical)`, AMPS will make a request to the entitlement module for `/parts/mechanical` when there is a message to deliver for that topic, and will make a separate request for `/parts/electrical` when there is a message to deliver for that topic.

Content Filtered Entitlements

The entitlement system offers the ability to enforce content restrictions on subscriptions. When AMPS requests `read` access to a `topic`, the module that performs entitlement can also return a filter to AMPS. This filter is evaluated independently of any filter on the subscription, and messages must match both the subscription filter and the filter provided by the entitlement to be returned to the application. If a message does not match the entitlement filter, the message is not delivered, regardless of whether the message matches the filters provided by the application.

AMPS also offers the ability to enforce content restrictions on `publish` commands. When AMPS requests `write` access to a `topic`, the module that performs entitlement can return a filter to AMPS. This filter is then evaluated against messages published to that topic by that user. If the message being published matches the filter, AMPS allows the message. Otherwise, AMPS rejects the message.

Message Queues

Message queues, since they are implemented as views over topics in the transaction log, present a special situation for the AMPS entitlement system in two ways. First, receiving a message from a queue implies that the subscriber has the ability to modify the contents of the queue. Second, a queue can specify an `DefaultPublishTopic` to receive publishes.

The AMPS entitlement system treats queues differently than other topics as follows:

- `read` entitlement on a queue also grants a user the ability to delete messages from the queue that are leased to that user. No other write permissions are implied.
- `write` entitlement on a queue grants the ability to publish to the queue, even in cases where AMPS translates that publish to the `DefaultPublishTopic` configured for the queue. No other permissions are implied. In particular, granting the `write` entitlement on a queue does not grant any entitlements on the `DefaultPublishTopic` directly: even though the message is delivered to the `DefaultPublishTopic`, the `publish` command must publish to the queue.

In all other respects, entitlements for message queues behave in the same way as entitlements for any other topic.

27.3. Providing an Identity for Outbound Connections (Authenticator)

For outgoing replication connections, AMPS may need to provide an identity and credentials to the replication destination. AMPS uses a module type called an *authenticator* to provide those credentials and handle any challenge/response protocol required by the authentication module in the remote system.

AMPS provides a default authenticator module, `amps-default-authenticator-module`, that is automatically configured as the Authenticator for the instance if no other instance Authenticator is provided. This module provides a user name with no password. To determine the user provided to AMPS, the module uses the value of the User option to the module if one is provided. Otherwise, the module uses the current user of the AMPS process: if the current user cannot be determined by the system, the module falls back to the value of the `USER` environment variable..

The Authenticator used for a replication Destination must provide credentials that are accepted by the Transport of the remote instance that the Destination is connecting to. See the *AMPS Configuration Reference* for information on configuring the Authenticator for a Destination.

27.4. Protecting Data in Transit Using SSL

AMPS provides the ability to use Secure Sockets Layer (SSL) connections for communication with AMPS clients. See SSL Connections, the *AMPS Configuration Reference*, and the documentation for the AMPS clients for details. *In this release, SSL support is provided as a preview.*

AMPS uses SSL to encrypt network traffic between clients and servers. No information about the transport is passed to the AMPS authentication and entitlement system. Encryption at the network level is completely independent of the AMPS authentication and entitlement system, and these features can be used independently.

Chapter 28. Troubleshooting AMPS

This chapter presents common techniques for troubleshooting AMPS. Additional troubleshooting information and answers to common questions about AMPS are included on our support site at http://support.crankuptheamps.com/hc.

28.1. Planning for Troubleshooting

There are several steps that you can take before you need to troubleshoot a problem that will make troubleshooting easier. 60East recommends that you consider taking the following steps for a production instance of AMPS:

1. Configure the instance to log messages of at least `warning` or higher level. Some problems require more information, so increasing the amount of logging may make troubleshooting easier, if your instance has storage available.

2. Ensure that client applications use unique names. Wherever possible, ensure that those names can easily be traced back to the instance of the application. For example, you might use the name of application combined with the name of the logged on user as a unique name. This will help you to more quickly find log messages related to a problem.

3. Enable the administrative server. The administrative console is a good way to get a snapshot of the current state of a running instance.

4. If you are using replication, ensure that your AMPS instances have unique names. Where possible, use names that make it easy to relate replication messages to the servers that process the message. For example, you might relate the AMPS instance name to the purpose that the instance serves, the physical server that the instance runs on, or both.

5. Learn what normal operation looks like for your application. If possible, take the time to inspect the AMPS logs and the output of the administrator console when everything is working as expected. Applications vary in how they use AMPS, and what is normal for your application might indicate a problem in a different application. For example, if your application normally has a few publishers and many subscribers, seeing dozens of publishers come online may indicate that an application has unexpectedly started more publishers. Likewise, if no publishers are online, that may indicate an issue with connectivity to the AMPS server. Understanding normal behavior will help you to more easily and accurately spot problems.

28.2. Finding Information in the Log

The AMPS log is one of the most useful places to find information when there's a problem with your application. Here are some techniques to use for finding relevant information in the log.

- Ensure the log is capturing information that will be useful for diagnosing the problem. To detect a problem, 60East recommends logging at `warning` level and above. To fully troubleshoot an error, it may be necessary to log at `trace` level to see the exact behavior in AMPS.

- To find log messages that may indicate a problem, use the Linux grep tool to find log messages at warning, error, critical, or emergency levels. For example, you might use the following command line:

```
grep -E 'warning|error|critical|emergency' log_file
```

This will show lines from the log that contain messages logged at those levels. The text that AMPS uses for log messages is guaranteed not to include strings that duplicate one of the log levels, although information that you configure (such as client names, topic names, and so on) may contain those strings.

- If you know the name of the client that experienced the problem, you can use that name to get information about the client. It's often helpful to get log messages that include the client name and several lines of output after the client name to help you understand the context in which AMPS produced the message for the client name. To do this, you might you use the following command line:

  ```
  grep -B2 -A15 client_name log_file
  ```

  This command line looks for all occurances of the *client_name* in the log file, and prints two lines of context before the line that contains the client name, and ten lines of context after the line that contains the client name.

Once you've found the information you're looking for, the `ampserr` utility can help you look up more information on messages, as described in Section 20.9.

28.3. Reading Replication Log Messages

For replication connections, the replication source creates a client name that it uses to connect to the downstream instance. This client name contains the source, destination, sync setting, and protocol for the connection. The client name uses the following format:

```
source!destination!sync_setting!protocol
```

Notice, however, that this is a *client name*. The client name is the name used for the connection, but it does not indicate the direction of any particular message. As an example, consider a client name of:

```
OrderServer!HotBackup!sync!amps-replication
```

This client name is used for a connection that the AMPS instance named *OrderServer* has made to AMPS instance named *HotBackup*. The connection uses the amps-replication protocol, and was configured for synchronous replication at the time the client connected. In this case, a message like the following:

```
12-1002 client[OrderServer!HotBackup!sync!amps-replication] replication ack
 received: publish ack
      [txid=35922]
```

Means that a publish acknowledgement was received on the connection that *OrderServer* made to *HotBackup*.

28.4. Troubleshooting Disconnected Clients

One common symptom of problems in an AMPS application is that AMPS disconnects clients unexpectedly. AMPS disconnects clients in the following situations:

- When transaction logging is configured for the instance and a client with a duplicate name logs on
- When heartbeating is enabled, and the client misses a heartbeat

- When a slow client falls behind by more than the configured threshold
- When the entitlement cache for an instance is reset
- When the administration console disconnects a client
- When the transport is disabled

This section presents techniques to help you identify why clients are disconnected and correct any problems that may exist.

Locating the Reason for Disconnection

To discover the reason that a client was disconnected, use the following command to find the client name in the logs:

```
grep -B2 -A5 client_name log_file
```

The results of this can provide information as to why the client was disconnected. AMPS logs a reason for the disconnection if the disconnection was the result of an internal action by AMPS. If the disconnection was the result of an action from the Admin console, or the client chose to disconnect, the disconnection is logged, but no further information is given.

Duplicate Client Name Disconnection

When a client is disconnected due to another client with the same name logging on, the messages produced might look like:

```
2014-11-20T16:26:59.6408410-08:00 [5] warning: 02-0025 A client logon with
 an 'in use' client name for the same user id forced a disconnect of client:
 client[my-name] with user id:
```

To resolve this issue, ensure that clients use unique names when connecting to instances that configure a transaction log.

Missed Heartbeat Disconnection

When AMPS disconnects a client due to the client failing to heartbeat, the log messages produced look like the following:

```
2014-11-20T16:35:23.9185690-08:00 [6] error: 07-0042 AMPS heartbeat manager
 is disconnecting an unresponsive client: no-heartbeat-client
```

This error most often arises from severe network congestion, a deadlock or similar problem in the application that is preventing the AMPS client library from producing heartbeats, or a problem in AMPS that prevents AMPS from servicing heartbeat requests.

Slow Client Disconnection

The following shows sample log entries for slow client disconnection. If a client named `sleepy-client` was disconnected for being a slow client, the relevant entries in the transaction log might look like:

```
2014-11-20T15:33:06.8496430-08:00 [7] warning: 70-0011 client[sleepy-client]
 slow consumption detected, offline messages.
2014-11-20T15:33:06.8498130-08:00 [7] error: 70-0004 client[sleepy-client]
 is not consuming messages, disconnecting slow client
```

Notice that there may be a considerable period of time between the client being offlined and the client being disconnected.

There are several approaches to solving the problem:

- *Reduce the number of messages returned*. Clients most often fall behind when a SOW query or a replay from the transaction log returns a large number of messages. If possible, use content filtering to return a more precise set of messages.
- *Improve the rate at which the client handles messages*. If the client message handler takes a relatively long time to process the message, moving message processing onto a different thread or streamlining the processing may improve the speed of the client and allow the client to keep up.
- *Adjust the client offlining threshold*. You can also increase the number of messages that AMPS will buffer for a specific client, as described in the section called “Slow Client Management ”.

Admin Console Client Disconnection

Disconnection from the admin console provides no additional information, and produces a log message like the following:

```
2014-11-20T15:33:06.8502350-08:00 [4] info: 07-0013 client[sleepy-client]
 disconnected.
```

Admin Console Transport Disabled

A transport being disabled through the admin console produces messages like the following:

```
2014-11-20T16:04:00.9548130-08:00 [10] info: 07-0047 Transport[json-tcp]
 being disabled.
2014-11-20T16:04:00.9550150-08:00 [4] info: 07-0013 client[amps-json-tcp-18]
 disconnected.
```

Part IV. Building Applications with AMPS

Chapter 29. Sample Use Cases

To further your understanding of AMPS, we provide some sample use cases that highlight how multiple AMPS features can be leveraged in larger messaging solutions. For example, AMPS is often used as a back-end persistent data store for client desktop applications.

The provided use case shows how a client application can use the AMPS command `sow_and_suscribe` to populate an order table that is continually kept up-to-date. To limit redundant data from being sent to the GUI, we show how you can use a delta subscription command. You will also see how to improve performance and protect the GUI from over-subscription by using the `TopN` query limiter along with a `stats` acknowledgement.

29.1. View Server Use Case

Many AMPS deployments are used as the back-end persistent store for desktop GUI applications. Many of the features covered in previous chapters are unique to AMPS and make it well suited for this task. In this example AMPS will be act as a data store for an application with the following requirements:

- allow users to query current order-state (SOW query)
- continually keep the returned data up to date by applying incremental changes (subscribe)

For purposes of highlighting the functionality unique to AMPS, we'll skip most of the details and challenges of GUI development.

Setup

For this example, let's configure AMPS to persist FIX messages to the topic ORDERS. We use a separate application to acquire the FIX messages from the market (or other data source) and publish them into AMPS. AMPS accumulates all of the orders in its SOW persistence, making the data available for the GUI clients to consume.

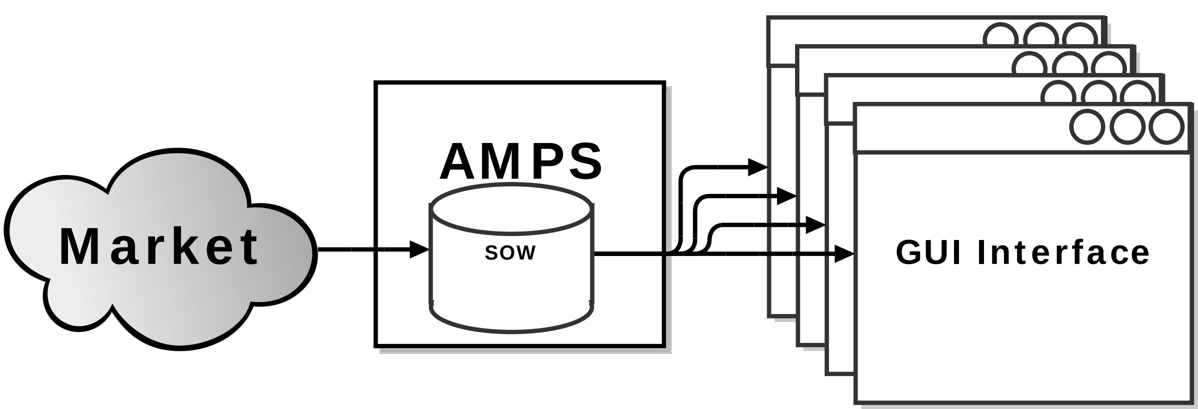


Figure 29.1. AMPS View Server Deployment Configuration

SOW Query and Subscription

The GUI will enable a user to enter a query and submit it to AMPS. If the query filter is valid, then the GUI displays the results in a table or “grid” and continually applies changes as they are published from AMPS to the GUI. For example, if the user wants to display active orders for `Client-A`, then they may use a query similar to this:

```
/11 = 'Client-A' AND /39 IN (0, 'A')
```

This filter matches all orders for `Client-A` that have FIX tag 39 (the FIX order status field) as 0 (’New’) or ’A’ (’Pending New’).

From a GUI client, we want to first issue a query to pull back all current orders and, at the same time, place a subscription to get future updates and new orders. AMPS provides the `sow_and_subscribe` command for this purpose.

> A more realistic scenario may involve a GUI Client with multiple tables, each subscribing with a different AMPS filter, and all of these subscriptions being managed in a single GUI Client. A single connection to AMPS can be used to service many active subscriptions if the subscription identifiers are chosen such that they can be demultiplexed during consumption.

The GUI issues the `sow_and_subscribe` command, specifying a topic of `ORDERS` and possibly other filter criteria to further narrow down the query results. Once the `sow_and_subscribe` command has been received by AMPS, the query returns to the GUI all messages in the SOW that, at the moment, match the topic and content filter. Simultaneously, a subscription is placed to guarantee that any messages not included in the initial query result will be sent after the query result.

The GUI client then receives a `group_begin` message from AMPS, signaling the beginning of a set of records returned as a result of the query. Upon receiving the initial SOW query result, this GUI inserts the returned records into the table, as shown in Figure 29.2. Every record in the query will have assigned to it a unique `SowKey` that can be used for future updates.

The receipt of the `group_end` message serves as a notification to the GUI that AMPS has reached the end of the initial query results and going forward all messages from the subscription will be live updates.

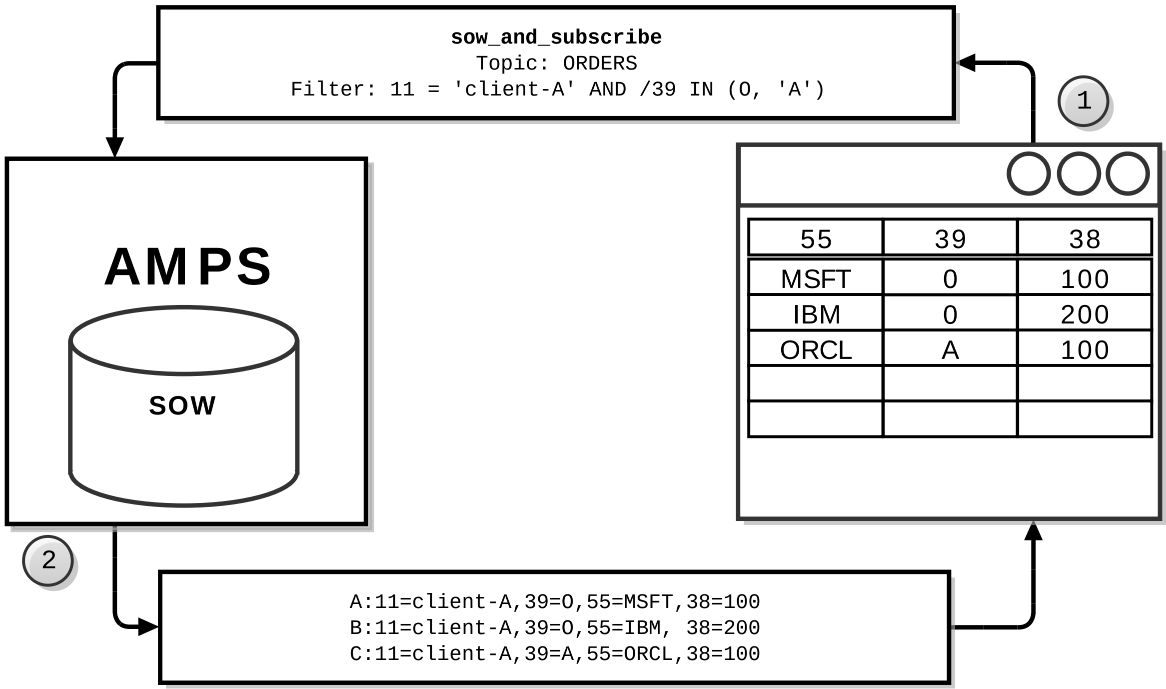


Figure 29.2. AMPS GUI Instance With sow_and_subscribe

Once the initial SOW query has completed, each `publish` message received by the GUI will be either a new record or an update to an existing record. The `SowKey` sent as part of each publish message is used to determine if the newly published record is an update or a new record. If the `SowKey` matches an existing record in the GUI's order table, then it is considered an update and should replace the existing value. Otherwise, the record is considered to be a new record and can be inserted directly into the order table.

For example, assume there is an update to order C that changes the order status (tag 39) of the client's `ORCL` order from 'A' to 0. This is shown below in Figure 29.3

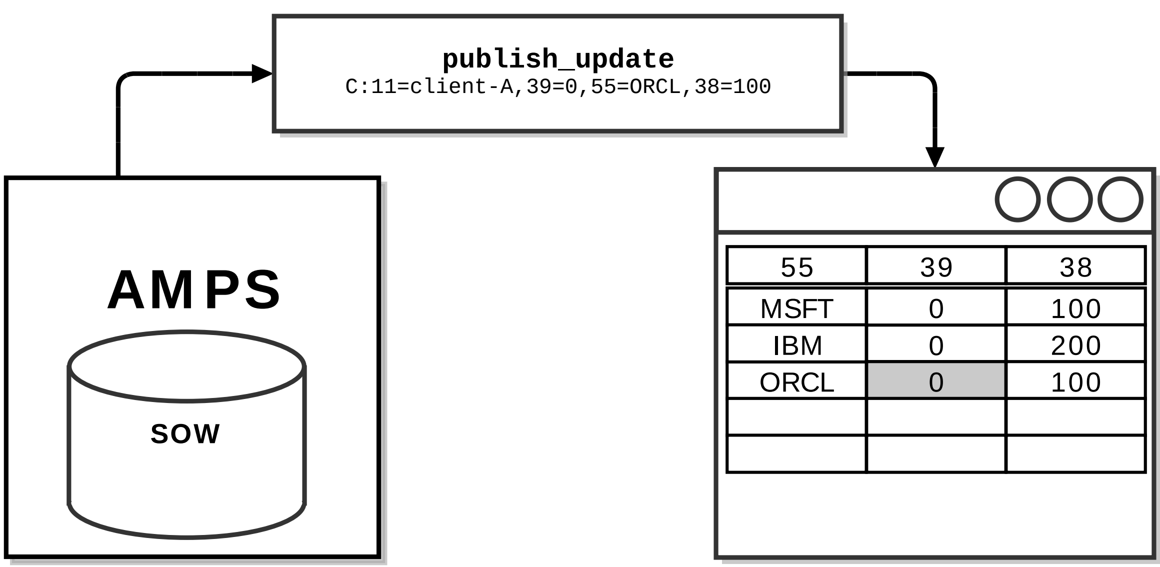


Figure 29.3. AMPS Mesage Publish Update

Out-of-Focus (OOF) Processing

Let's take another look at the original filter used to subscribe to the `ORDERS` SOW topic. A unique case exists if an update occurs in which an `ORDER` record status gets changed to a value other than 0 or 'A'. One of the key features of AMPS is OOF processing, which ensures that client data is continually kept up-to-date. OOF processing is the AMPS method of notifying a client that a new message has caused a SOW record's state to change, thus informing the client that a message which previously matched their filter criteria no longer matches or was deleted. For more information about OOF processing, see Chapter 10.

When such a scenario occurs, AMPS won't send the update over a normal subscription. If OOF processing is enabled within AMPS by specifying the `oof` option for this subscription, then updates will occur when previously matching records no longer match due to an update, expiration, or deletion.

For example, let's say the order for `MSFT` has been filled in the market and the update comes into AMPS. AMPS won't send the published message to the GUI because the order no longer matches the subscription filter; AMPS instead sends it as part of an OOF message. This happens because AMPS knows that the previous matching record was sent to the GUI client prior to the update. Once an OOF message is received, the GUI can remove the corresponding order from the orders table to ensure that users see only the up-to-date state of the orders which match their filter.

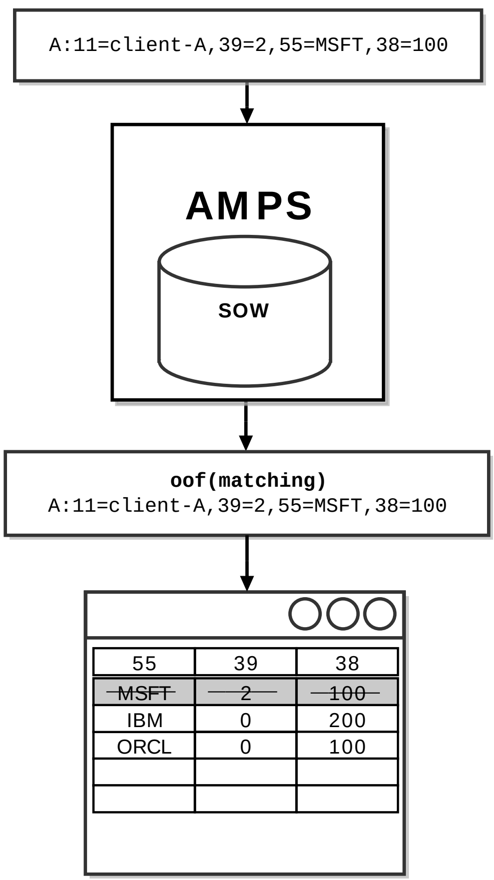


Figure 29.4. AMPS OOF Processing

Conclusion and Next Steps

In summary, we have shown how a GUI application can use the `sow_and_subscribe` command to populate an order table, which is then continually kept up-to-date. AMPS can create further enhancements, such as those described below, that improve performance and add greater value to a GUI client implementation.

sow_and_delta subscribe

The first improvement that we can make is to limit redundant data being sent to the GUI, by placing a `sow_and_delta_subscribe` command instead of a `sow_and_subscribe` command. The `sow_and_delta_subscribe` command, which works with the FIX and NVFIX message types, can greatly reduce network congestion as well as decrease parsing time on the GUI client, yielding a more responsive end-user experience.

With a delta subscription, AMPS Figure 29.3 sends to the subscriber only the values that have changed: `C:39=0` instead of all of the fields that were already sent to the client during the initial SOW query result. This may seem to make little difference in a single GUI deployment; but it can make a significant difference in an AMPS deployment with hundreds of connected GUI clients that may be running on a congested network or WAN.

TopN and Stats

We can also improve client-side data utilization and performance by using a `TopN` query limiter with a stats acknowledgment, which protects the GUI from over-subscription.

For example, we may want to put a 10,000 record limit on the initial query response, given that users rarely want to view the real-time order state for such a large set. If a `TopN` value of 10000 and an `AckType` of `stats` is used when placing the initial `sow_and_subscribe` command, then the GUI client would expect to receive up to 10,000 records in the query result, followed by a `stats` acknowledgment.

The `stats` acknowledgement is useful for tracking how many records matched and how many were sent. The GUI client can leverage the `stats` acknowledgment metrics to provide a helpful error to the user. For example, in a scenario where a query matched 130,000 messages, the GUI client can notify the user that they may want to refine their content filter to be more selective.

Part V. Appendices

Appendix A. AMPS Distribution Layout

This appendix lists layout of the AMPS distribution, with special focus on the binaries present in the layout. Use this appendix to plan your AMPS deployment.

60East recommends that all AMPS deployments contain the full contents of the `/bin` and `/lib` directories. For development installations that are extending the AMPS server, your installation should contain the `/api` and `/sdk` directories (as well as the *AMPS Server SDK*, available as a separate download from the 60East web site).

The AMPS distribution contains the following items at the top level:

Table A.1. AMPS Distribution Contents

Item	Description
`/bin`	AMPS binaries: the AMPS server, daemon deployment scripts, AMPS utilities, and `spark`.
`/docs`	AMPS base documentation. Current versions of the documentation and additional guides are available from the 60East website.
HISTORY	Revision history for AMPS releases, containing information on changes for each version of AMPS.
`/lib`	Libraries used by the AMPS binary.
LICENSE	The AMPS license.
README	The README file for AMPS.
`/sdk`	Headers used for modules that extend AMPS.

A.1. /bin directory

Table A.2. AMPS /bin directory Contents

Item	Description
`amps_bio_perf_test`	Diagnostic tool for testing the performance of I/O systems.
`amps_client_ack_dump`	Utility for showing the contents of the AMPS client.ack file, containing persistent per-client information.
`ampserr`	Utility for looking up details on AMPS log file items.
`ampServer`	The AMPS server binary.
`ampServer-compat`	The downward compatible version of the AMPS server binary. This version avoids using some of the hardware capabilities present in newer CPU architectures.
`amps_file`	A utility for identifying the type of AMPS files and the file format that the file uses.
`amps-init-script`	Part of the AMPS service installation. This script is installed into the init.d directory when the AMPS service is installed.

Item	Description
`amps_journal_dump`	Utility for extracting the contents of AMPS transaction log journal files.
`amps_mt_perf_test`	Diagnostic tool for performance testing of the AMPS engine parsing infrastructure.
`amps_sow_dump`	Utility for extracting the contents of AMPS SOW files.
`amps-sqlite3`	Convenience wrapper for querying an AMPS statistics database.
`amps_upgrade`	Utility for upgrading data files from previous versions of AMPS to the current version.
`install-amps-daemon.sh`	Installation script for installing AMPS as a Linux service.
`/lib`	Directory containing the libraries used by the `spark` utility.
`spark`	Utility that provides a command-line interface to AMPS.
`uninstall-amps-daemon.sh`	Installation script for removing the AMPS Linux service from the system.

Appendix B. Configuration File Shortcuts

This appendix describes features that AMPS provides for simplifying configuration files.

B.1. AMPS Configuration File Special Characters

In AMPS there are a few special characters that you should be aware of when creating your configuration file. These characters can provide some handy short cuts and make configuration creation easier, but you should also be aware of them so as not to introduce errors.

State of the World File Name

When specifying the file for a State of the World database, using the `%n` string in the file name specifies that the AMPS server will use the message type and topic name in that position to create a unique filename. Example B.1 shows how to use this in the AMPS configuration file.

```
 <SOW>
    <Topic>
      <Topic>Customers</Topic>
      <FileName>./sow/%n.sow</FileName>
      <MessageType>json</MessageType>
      <Key>/customerId</Key>
    </Topic>
</SOW>
```

Example B.1. SOW file name tokens used in configuration file

Log Rotation Name

When specifying an AMPS log file which has `RotationThreshold` specified, using the `%n` string in the log file name is a useful mechanism for ensuring the name of the log file is unique and sequential. Example B.2 shows a file name token replacement in the AMPS configuration file.

```
<Logging>
  <Target>
    <Protocol>file</Protocol>
    <Level>info</Level>
    <FileName>log/log-%n.log</FileName>
    <RotationThreshold>2G</RotationThreshold>
  </Target>
</Logging>
```

Example B.2. Log file name tokens used in configuration file

In the above example, a log file will be created in the `AMPSDIR/log/` directory. The first time this file is created, it will be named `log-1.log`. Once the log file reaches the `RotationThreshold` limit of 2G, the previous log file will be saved, and the new log file name will be incremented by one. Thus, the next log file will be named `AMPSDIR/log/log-2.log`.

Dates

AMPS allows administrators to use date-based file names when specifying the file name in the configuration, as demonstrated in Example B.3.

```
<Logging>
  <Target>
    <Protocol>file</Protocol>
    <Level>info</Level>
    <FileName>
      log/log-%Y-%m-%dT%H%M%S.log
    </FileName>
    <RotationThreshold>2G</RotationThreshold>
  </Target>
</Logging>
```

Example B.3. Date tokens used in configuration file

In the above example, a log file will be created in the `$AMPSDIR/log` named `2011-01-01-120000.log` if the log was created at noon on January 1, 2011.

AMPS provides full support for the date tokens provided by the standard strftime function, with the exception of `%n`, as described above. The following table shows some of the most commonly used tokens:

Table B.1. Commonly Used Date and Time Tokens

Token	Provides	Example
%a	Short weekday name	Fri
%A	Full weekday name	Friday
%b	Short month name	Feb
%B	Full month name	February
%c	Simple date and time	Fri Feb 14 17:25:00 2014
%C	Century	20
%d	Day of the month (leading zero if necessary)	05
%D	Short date format (MM/DD/YY)	02/20/14
%e	Day of the month (leading space if necessary)	5
%F	Short date format (YYYY-MM-DD)	2014-02-20
%H	Hour (00-23)	17
%I	Hour (00-12)	05
%j	Day of the year (001-366)	051
%m	Month (01-12)	02
%p	AM or PM	PM
%r	Current time, 12 hour format	05:25:00 pm

Token	Provides	Example
%R	Current time, 24 hour format	17:25
%T	ISO 8601 Time format	17:25:00
%u	ISO 8601 day of the week (1-7, Monday = 1)	5
%V	ISO 8601 week number (00-53)	07
%y	Year, last two digits	14
%Y	Year, four digits	2014
%Z	Timezone name or abbreviation (blank if undetermined)	PST

B.2. Using Units in the Configuration

To make configuration easy, AMPS permits the use of units to expand values. For example, if a time interval is measured in seconds, then the letter `s` can be appended to the value. For example, the following SOW topic definition used the `Expiration` tag to set the record expiration to 86400 seconds (one day).

```
<SOW>
  <Topic>
    ...
    <Expiration>86400s </Expiration>
    ...
  </Topic>
</SOW>
```

Example B.4. Expiration Using Seconds

An even easier way to specify an expiration of one day is to use the following `Expiration`:

```
<SOW>
  <Topic>
     ...
     <Expiration>1d</Expiration>
     ...
  </Topic>
</SOW>
```

Example B.5. Expiration Using Days

Table B.2 shows a listing of the time units AMPS supports in the configuration file.

Table B.2. AMPS Configuration - Time Units

Units	Description
`ns`	nanoseconds
`us`	microseconds
`ms`	milliseconds
`s`	seconds
`m`	minutes
`h`	hours

Units	Description
d	days
w	weeks

AMPS configuration supports a similar mechanism for byte-based units when specifying sizes in the configuration file. Table B.3 shows a listing of the byte units AMPS supports in the configuration file.

Table B.3. AMPS Configuration - Byte Units

Units	Description
kb	kilobytes
mb	megabytes
gb	gigabytes
tb	terabytes

Dealing with large numbers in AMPS configuration can also be simplified by using common exponent values to handle raw values. This means that instead of having to input `10000000` to represent ten million, a user can input `10M`. Table B.4 contains a list of the exponents supported.

Table B.4. AMPS Configuration - Numeric Units

Units	Description
k	10^3 - thousand
M	10^6 - million

To make it easier for users to remember the units, AMPS interval and byte units are not case sensitive.

B.3. Environment Variables in AMPS Configuration

AMPS configuration also allows for environment variables to be used as part of the data when specifying a configuration file.

If a global system variable is commonly used in an organization, then it may be useful to define this in one location and re-use it across multiple AMPS installations or applications. AMPS will replace any token wrapped in `${}` with the environment variable defined in the current user operating system environment. Example B.6 demonstrates how the environment variable `ENV_LOG` is used to define a global environment variable for the location of the host logging.

```
<Logging>
  <Target>
    <Protocol>file</Protocol>
    <FileName>${ENV_LOG}</FileName>
    <Level>info</Level>
    <RotationThreshold>2G</RotationThreshold>
  </Target>
</Logging>
```

Example B.6. Environment Variable Used in Configuration

Internal Environment Variables

In addition to supporting custom environment variables, AMPS includes a configuration variable, AMPS_CONFIG_DIRECTORY, which can be used to reference the directory in which the configuration file used to start AMPS is located. For example, assume that AMPS was started with the following command at the command prompt:

```
%>./ampServer ../amps/config/config.xml
```

Given this command, the log file configuration option shown in Example B.7 can be used to instruct AMPS to create the log files in the same parent directory as the configuration file — in this case ../amps/config/logs/infoLog.log.

```
<Logging>
  <Target>
    <Protocol>file</Protocol>
    <FileName>
      ${AMPS_CONFIG_DIRECTORY}/logs/infoLog.log
    </FileName>
    <Level>info</Level>
    <RotationThreshold>2G</RotationThreshold>
  </Target>
</Logging>
```

Example B.7. AMPS_CONFIG_DIRECTORY Environment Variable Example

In addition to the AMPS_CONFIG_DIRECTORY environment variable, AMPS also supports the AMPS_CONFIG_PATH, which is an absolute path to the configuration file used to start AMPS.

Glossary of AMPS Terminology

acknowledgement	a networking technique in which the receiver of a message is responsible for informing the sender that the message was received
conflated topic	a copy of a SOW topic that conflates updates on a specified interval. This helps to conserve bandwidth and processing resources for subscribers to the conflated topic.
conflation	the process of merging a group of messages into a single message. e.g. when sending acknowledgment messages for a group of sequential messages, sending only the most recent message can be used to conflate all messages which have outstanding acknowledgments waiting to be processed.
filter	a text string that is used to match a subset of messages from a larger set of messages.
message expiration	the process where the life span of records stored are allowed limited.
message type	the data format used to encapsulate messages
oof (out of focus)	the process of notifying subscribing clients that a message which was previously a result of a SOW or a SOW subscribe filter result has either expired, been deleted from the SOW or has been updated such that it no longer matches the filter criteria.
replication	the process of duplicating the messages stored into an AMPS instance for the purpose of enabling high availability features.
replication source	an instance of AMPS which is the primary recipient of a published message which are then sent out to a replication destination.
replication destination	the recipient of replicated messages from the replication source.
slow client	a client that is over-subscribed and being sent messages at a rate which is faster than it can consume.
SOW (State of the World)	the last value cache used to store the current state of messages belonging to a topic.
topic	a label which is affixed to every message by a publisher which used to aggregate and group messages.
transport	the network protocol used to to transfer messages between AMPS subscribers, publishers and replicas.
transaction log	a history of all messages published which can be used to recreate an up to date state of all messages processed.
view	a data relation which is constructed from the records of a SOW topic.

Index

Symbols

A

B

C

D

E