Highly Available AMPS Client Programming

60East Technologies

Jun 26, 2017

1. Overview

Building a highly available distributed system is hard. 60East AMPS technology greatly simplifies the chore when you use AMPS for message passing and filtering between nodes in your system. You can configure AMPS instances to replicate each other, allowing you to build a network of AMPS instances that will survive both network and system outages.

To enable high availability throughout your entire system, however, your clients may need to know how to gracefully detect and recover from outages. This white paper discusses common failure modes and recovery options for all kinds of AMPS clients, and discusses support available in the AMPS Client Library.

This paper discusses a replication architecture that works with all versions of AMPS that support replication. However, starting with AMPS 3.8.0.0, AMPS supports passthrough replication and replication groups, which greatly simplify replication. The instructions in this whitepaper will still work with AMPS 3.8.0.0, but are not the most efficient way to configure replication. Full details on replicaiton in 3.8.0.0 and later are available in the User's Guide for those versions.

2. Recovering from Connectivity Outages

One of the most basic problems in high availability communications is disconnection between distributed endpoints. An AMPS client, for example, may disconnect from a remote server for a variety of reasons: a physically disconnected network segment, a malfunctioning or misconfigured firewall or router, or even a down AMPS server. As a starting point, highly available AMPS clients need to reconnect to their existing server when a disconnection occurs.

Disconnect Detection

In most cases, you become aware that your connection is no longer alive when the TCP stack on your client system reports a failure to your application. With the AMPS Client Library, disconnections that are detected and reported by the operating system result in an invocation of your disconnect handler. In AMPS, calling the `setDisconnectHandler()` method registers a disconnect handler that is called when a disconnection occurs:

```
class MyDisconnectHandler implements
  ClientDisconnectHandler
{
```

```
  public void invoke(Client c)
  {
      c.connect(...); // try to reconnect to a server
  }
}

Client c = new Client(...);
c.setDisconnectHandler(new MyDisconnectHandler());
```

Example 1. Disconnect Handling

It can be difficult for your application to detect disconnection in a timely fashion. For example, unless your client uses the heartbeat mechanism, a publisher or subscriber may remain running for minutes with an invalid connection before the operating system reports that the connection is not valid.

The operating system may not notify the client that an idle socket has been closed until an extended period of time has elapsed, or the client attempts to send data to the socket. Current AMPS servers and clients provide a heartbeat mechanism for reducing the amount of elapsed time before detecting that the connection is no longer active.

Choosing a Client Name

In AMPS, each client must log on with a unique client name. An AMPS instance will not allow multiple connections to log on with the same client name. When publishing, AMPS uses the client name as a key to record the highest sequence number seen from each client; when a publisher disconnects and reconnects to the same AMPS instance or a secondary, it should log on with the same client name as it used previously.

AMPS will disconnect a client if it attempts to log on with a client name that is in use by another connection.

With the AMPS Client Library, applications pass the client name to the Client constructor. If you are using the `HAClient` interface, pass the client name in the static methods `createFileBacked` or `createMemoryBacked`.

Choosing a Server

Since disconnection can occur for so many reasons, an application needs to be smart about choosing a new server. In the case of disconnection with a single AMPS instance, the only choice is to attempt reconnection to the same server, or to give up entirely. You face a more complicated situation, however, when building a system with a primary and secondary; the application's options are even more complex for choosing a new server when your system involves a mix of backup servers in the same rack and others in entirely different data centers.

As a rule, you should always attempt to reconnect to the *primary* when a disconnection occurs. Network "hiccups" happen even when both the client and server hosts are healthy. Choose to move on to secondary servers only when reconnection to the primary fails. In most cases, your code should prefer a secondary that is close by rather than far away; but it is important to keep this logic as simple as possible, since the code for choosing a new server will be among the hardest in your application to fully test.

The AMPS Client Library allows you to encapsulate your logic for choosing a server and reuse it across multiple applications, by implementing the `ServerChooser` interface and using it with the `HAClient`. You provide the

`ServerChooser`, both to establish an initial connection and to choose an alternate connection. The `HAClient` works by implementing a disconnect handler for you, and by interacting with the `ServerChooser`. The reference implementation, `DefaultServerChooser`, keeps a list of URIs to connect with; upon an initial disconnection, it attempts to reconnect to the previously connected server, and if that fails, it keeps trying alternates in its list. You can use this reference implementation out of the box, extend it for your own needs, or create your own `ServerChooser` from scratch.

Here is an example using the reference implementation:

```
HAClient haClient = HAClient.createMemoryBacked(...);

  DefaultServerChooser myServerList = new DefaultServerChooser();
  myServerList.add("tcp://primary.amps:9004/fix");
  myServerList.add("tcp://seconday.amps:10000/fix");
  myServerList.add("tcp://far.away.amps:9004/fix");

  haClient.setServerChooser(myServerList);

  haClient.connectAndLogon();

  // After this point, the haClient will
  // automatically atttempt reconnection
  // and connection to the secondary servers
  // provided by "myServerList".
```

Example 2. HA Client - Server Chooser

Implementing your own `ServerChooser` can create a distinct advantage since you will be able to incorporate knowledge about the particulars of the local network environment. In large organizations with multiple AMPS deployments, a central team may be responsible for implementing a custom ServerChooser that utilizes proprietary knowledge to provide even greater reliability and efficiency than the `DefaultServerChooser`. Contact your site's AMPS team for more details.

Establishing a new connection after connectivity is broken is an important part of creating highly available AMPS clients in nearly every scenario. Though AMPS does not segregate workloads, most AMPS clients are primarily either publishers or subscribers. For the remainder of this white paper, we will focus first on issues specific to publishers, followed by issues specific to subscribers.

3. Reliable Publishing

A reliable publisher must ensure that it actually succeeds in publishing the messages that it intends to publish. Before talking about ways to ensure publication, we will review a few concepts that are covered in more detail in the AMPS User's Guide.

Publishing Acknowledgment

In a non-HA publish, a publisher simply sends a `publish` message to an AMPS server; the AMPS server does not send an acknowledgement ("ACK") back to the client. This mode of publishing allows for maximum performance,

but with no guarantees. What happens if the AMPS server does not receive your message? What if the AMPS server goes down before it can persist the message or send it on to subscribers?

In order to ensure that messages are seen by AMPS and do not need to be republished, a publisher can supply a sequence number on published messages. The publisher's sequence numbers uniquely identify each message, and enable the publisher to request a `persisted` ACK from the server indicating that AMPS has both received the message and persisted it in the transaction log. Sequence numbers are monotonically increasing integers, and must begin with the last value used by the client `+ 1`. Publishers are responsible for determining and maintaining this sequence. If a disconnect occurs before the publisher receives the server's acknowledgement, then the publisher must re-publish all of the unacknowledged messages on the new connection after reconnection occurs.

Once a publisher begins supplying sequence numbers on published messages, it must continue to do so for the remainder of its lifetime.

To maximize performance in this mode of publishing, AMPS does not send a separate ACK for each message published. Instead, AMPS sends periodic ACK message to a publisher containing the last sequence number that has been persisted; a client can safely use this and disregard all of the messages with sequence numbers less than or equal to this value.

During publisher recovery, AMPS will always return the last sequence number it has acknowledged in the previous connection, which is likely lower than the highest sequence number it has seen from that publisher. As a result, AMPS is likely to consider the first few replayed messages as duplicates.

In support of this mechanism, AMPS will inform a publisher about the last sequence number it has seen from that publisher when logging on. For example, if a publisher has published messages, 1, 2, 3 and 10, and then disconnected, AMPS will tell that publisher (upon a subsequent connection) that the sequence number is 10, so that the publisher can begin from the new value. Likewise, if upon reconnection the server informs the publisher that the most recent sequence number observed is 2, then the publisher is responsible for re-publishing 3 and 10 to the server.

Recovering from Disconnect or Server Failure

As noted above, AMPS expects a highly available publisher to be able to republish any messages for which AMPS has not yet sent an ACK. To fulfill this expectation, publishers must either store or be able to re-create previously published messages. To implement such a mechanism with reasonable performance, most publishers will need to follow these recommendations:

1. Keep an in-memory store of the contents of messages and their corresponding sequence numbers. Just before sending a message to the server, the publisher must calculate a new sequence number, and store it along with the data and topic. This code path will occur once per sent message, so its performance must be optimal.

2. When a persisted ACK arrives from the server, free up any space held in this store by messages whose sequence number is less than or equal to the one sent in the ACK. This code path will occur frequently as well, though typically less often than one per message (a ratio of 1:10 to 1:100 is expected, but AMPS provides no guarantees).

3. When logging on to the server (either in the case of an initial connection or a re-connection), a publisher should use the sequence number returned by AMPS to calculate a starting point for the sequence numbers on new messages. AMPS will immediately discard messages sent with sequence numbers less than or equal to the one supplied by the server on logon, considering them to be duplicates of previously received messages.

4. When recovering from a disconnection, if the server returns a sequence number less than or equal to the most recent message you published, you must re-publish each of the messages that the server has not yet seen (or else, they are lost forever).

If you built your publisher on the AMPS Client Library, all of this functionality exists in the MemoryPublishStore, which the client library creates for you when using a memory-backed HA client:

```
// Creates a memory-backed HA client that tracks
// bookmarks and subscriptions in memory.
MessageHandler handler = new MyHandler();
HAClient myClient = HAClient.createMemoryBacked("myClient");

// Add your instances to the client.
DefaultServerChooser chooser = new DefaultServerChooser();
chooser.add("tcp://ubuntu.local:9004/fix");
chooser.add("tcp://backupampsinstance:9004/fix");

myClient.setServerChooser(chooser);
myClient.connectAndLogon();

// These messages are all stored in memory
// while we await an ACK from AMPS.
for(int i = 0; i < 100; i++)
{
myClient.publish("someTopic",
String.format("12345=%d", i));
}
```

Example 3. HA Publisher Client

In this example, each of the 100 published messages are stored in memory until the AMPS client receives acknowledgement from the server that it has received and persisted them. If a disconnect and reconnect occurs, the AMPS client will automatically re-publish any messages that need to be re-published, based on the rules above. The AMPS client automatically calculates and manages the sequence number for published messages.

Safe Shutdown

When implementing a short-lived publisher, it is important not to shut down until you are sure that the AMPS server has acknowledged all messages that you have published. If the publisher shuts down before this, it runs the risk of message loss -- for example, the AMPS server might also shut down, and messages that it has not yet received or persisted may be lost for good.

Before shutting down, well-behaved publishers will examine the last ACK received from AMPS, to verify that it is equal to the sequence number on the last published message. If it is not, the publisher should simply wait; assuming the connection does not fail, AMPS will eventually send the final ACK, and you can proceed with shutting down. If you choose to implement a timeout mechanism, ensure that the publisher will attempt to reconnect after a timeout expiration and that it checks the returned sequence number in the logon, replaying any messages that the server has not yet seen.

To aid in this aspect of publisher implementation, the AMPS Client Library provides a facility to inquire how many unpersisted (unacknowledged) messages remain. You can choose to wait for this value to become 0 before exiting:

```
for(int i = 0; i < 100; i++)
{
```

```
  myClient.publish("someTopic", String.format("12345=%d", i));
}

// Just wait to hear back from the server.
while(myClient.getPublishStore().unpersistedCount() > 0)
{
  Thread.yield();
}

myClient.disconnect();
```

Example 4. HA Publisher checking for unacknowledged messages

Untimely Publisher Death

Your publisher may terminate before you expect it to. Perhaps, for example, another user kills your process, or your process terminates with a fatal error, or the operating system kills your process because it runs out of resources. In such scenarios, your publisher may exit before it has received acknowledgement of all of the messages it attempted to publish. Combined with a server failure at roughly the same time, this could result in lost messages.

To guard against losing messages when a publisher goes down, a publisher must persist messages outside its process, typically to disk. If a publisher exits, a new process can start, log on, and compare the sequence number returned by the server to the messages stored on disk. The publisher replays any messages that were stored with a sequence number higher than the one returned by the AMPS server.

Publishers that write their messages to disk before publishing have greater reliability, but lower performance. Efficient use of disk resources, both in terms of overall footprint and IOPS, are critical in implementing a high-performance publish store. If you wish to protect against host or operating system failure, an additional overhead of flushing buffers to disk must occur as well, further lowering performance.

The AMPS Client Library includes a default, extensible implementation of a disk-backed publish store in the `PublishStore` class. It is instantiated for you when using the `createFileBacked` method of `HAClient`:

```
// Creates a file-backed HA client that tracks
// bookmarks and subscriptions on disk
MessageHandler handler = new MyHandler();
HAClient myClient = HAClient.createFileBacked(
        "myClient", // client name
        "myClient.publish", // publish filename
        "myClient.subscribe" // subscribe filename
        ); //1

myClient.connectAndLogon(); //2

for(int i = 0; i < 100; i++)
{
myClient.publish("someTopic",
String.format("12345=%d", i));
}
```

Example 5. Memory-backed HA Publisher

In this example, the publisher writes each of the 100 messages to the `myClient.publish` file on disk before sending them to the server. If the publisher process terminates before the server can acknowledge the published messages, then the data of those messages remains on disk. When the publisher restarts, the `createFileBacked` method will reload the `myClient.publish` file (by, in turn, creating a `PublishStore` object), on the line marked `//1`. When the publisher successfully logs back on to a server (line `//2`), the AMPS client will automatically compare the sequence number returned by AMPS with the loaded contents of `myClient.publish`, and automatically re-publishes any messages that the server needs.

4. Subscriber Concerns

Once a client places a subscription with AMPS, the AMPS instance ensures that it sends every matching message to that client. If the subscriber loses its connection to AMPS, the subscriber will never see any of the messages that AMPS processed while the subscriber was away. To allow clients to resubscribe at the point where they originally lost connectivity, AMPS provides a bookmark mechanism whereby it stamps messages with a string (the bookmark) that uniquely identifies the message and its order relative to other messages AMPS has seen.

By default, AMPS saves network bandwidth by omitting the bookmark on messages that it sends to a subscriber. Subscribers that are interested in a bookmark must supply a bookmark when issuing their `subscribe` command, supplying one of the following as the bookmark:

- **A valid bookmark string**. Valid bookmark strings identify a message and publisher id. AMPS will begin the subscription at the point in the transaction log immediately following this bookmark.
- **The value `0`**. This special value represents the epoch. AMPS will begin the subscription with the first message in its transaction log.
- **The value `0|1|`**. AMPS interprets this value as a request to begin the subscription with the next message; that is, now. AMPS will not perform a replay of past messages.

To implement a client that does not lose messages upon reconnect, the client must retain the latest bookmarks it has seen for each publisher on each subscription and, when re-subscribing, supply a comma-delimited list of these bookmarks in the header field of the `subscribe` command. If an application requires resilience to subscriber failure, the application should keep the most recently seen bookmark for each subscription on disk. If performance is the greatest concern of the application and the ability to ensure no lost messages on subscriber failure is not required, then the most recently seen bookmark can simply be kept in memory.

Resubscribing after Disconnect

If disconnection occurs, the AMPS server does not automatically reinstate the subscriptions when your client reconnects. Each subscriber is responsible for reissuing subscriptions that it wishes to continue; and for bookmark subscriptions, these must include the bookmark at which the subscription should resume.

The AMPS Client Library provides a facility for remembering the subscriptions your application has made and reissuing them on reconnect in the `SubscriptionManager` interface. AMPS provides a default implementation in the `MemorySubscriptionManager` class, which the client automatically creates for you when you use the `HAClient`:

```
MessageHandler handler = new MyHandler();
HAClient myClient = HAClient.createMemoryBacked("myClient" // client name
```

```
  );

DefaultServerChooser chooser = new DefaultServerChooser();
chooser.add("tcp://localhost:9004/fix");

myClient.setServerChooser(chooser);
myClient.connectAndLogon();

myClient.subscribe(handler, "someTopic", 10000);

// someTopic will be resubscribed to, even
// if we disconnect (and reconnect).
```

Example 6. Subscribe with Reconnect

In this example, after the `subscribe()` call returns successfully, the `HAClient` will resubscribe to `someTopic` whenever a disconnection and subsequent reconnection occurs.

When using the `HAClient.createFileBacked()` method to create a file-backed HA client, the list of subscriptions is kept only in memory, and the AMPS client does not automatically re-create subscriptions when your process restarts.

If you would like to implement an application that is capable of automatically recreating an arbitrary set of subscriptions on start-up, then you should implement the `SubscriptionManager` interface, writing subscriptions to disk. On recovery of the file in the new subscriber process, your new `SubscriptionManager` must determine how to bind previous subscriptions to new `MessageHandler` instances, and then re-create the desired subscriptions on the new Client.

Message Receive Order versus Processing Order

Some subscribers need to process messages in a different order from which AMPS sends them. Imagine a subscriber that calculates statistics on a moving window of trades. At any given time, it needs to wait for the current window to fill up before calculating and issuing the new output; once it has computed and delivered the output, it can free up the messages and resources associated with that window. This hypothetical subscriber might need to have multiple windows open at any given time as well, because messages do not necessarily arrive in the order that real events occur.

If this hypothetical subscriber failed and restarted, it would need to re-subscribe to AMPS to get messages corresponding to all of the windows that were inprogress at the time it went down. It would be important not to lose any messages that occurred while the subscriber was down. It would also be important not to receive messages again for any windows that are already closed even though those messages might have originally been delivered after messages for windows that are open. In short, the application should not reprocess all messages after the re-subscribe point; only those messages that the application has not yet discarded should be reprocessed.

In this complex (but common) scenario, the order of message processing is different from the order of message reception. Clients seeking to adequately support this scenario must store both a bookmark to resubscribe to ”the earliest message that the application has not discarded” along with the bookmarks associated with the discarded messages after that point. After a resubscribe, the application should filter out incoming messages that have bookmarks matching those already discarded.

The AMPS Client Library provides built-in support for resubscribing to a point in time with filtering of already discarded messages through the `BookmarkStore` interface. The AMPS Client Library provides three implementations of the `BookmarkStore`:

- `MemoryBookmarkStore` implements an in-memory store that allows you to automatically re-subscribe where you left off, in case of a connection failure or server outage;
- `LoggedBookmarkStore` implements a disk-based store that logs a bookmark for each incoming message and each discarded message, in case of network, server, or subscriber failure;
- `RingBookmarkStore` implements a much faster disk-based store that does not filter discarded messages after the resubscribe point, thus potentially delivering duplicates if your application processes messages in a different order than it receives them.

When using a `BookmarkStore`, it is incumbent upon your application to call the store's `discard()` method when it is logically finished with a message, so that AMPS does not redeliver the message upon resubscribe:

```
public void invoke(Message message)
{
  ...
  myClient.getBookmarkStore().discard(
  message.getSubIdRaw(),     // The subscription ID
  message.getBookmarkSeqNo() // The sequence number of this message.
    );
  ...
}
```

Example 7. Bookmark Discard

A subscriber must call `discard()` as soon as possible after processing a message. If a subscriber uses a bookmark store but fails to call `discard()` on messages as they are processed, recovery will always result in duplicate messages.

Publisher Duplicates

It is possible for a subscriber to see duplicate messages from AMPS, even when it properly re-subscribes to the bookmark it last saw before disconnection. The following conditions are necessary for this situation to occur:

- The subscriber must have specified the live option when placing the subscription, indicating that AMPS should deliver matching messages before they are stored in the transaction log;
- There must be a failure of connectivity or an AMPS server that causes publishers and subscribers to fail over to a secondary.

If this scenario occurs, it is possible for publishers to replay a message that the new AMPS instance does not recognize as a duplicate, but that subscribers have already seen. It is therefore possible that the subscriber will, upon connection to the secondary and re-subscription, observe messages arriving from the new AMPS instance that were already delivered by the old instance, thus resulting in apparent duplicates.

To prevent this scenario, subscribers need to determine if an incoming bookmark is an unexpected duplicate of one already seen. Subscribers must parse the bookmark into the publisher ID and sequence number components, which are separated by the | (pipe) character. Subscribers should filter out messages with out-of-order sequence numbers from a given publisher ID. For example, if a subscriber receives messages from the same publisher with sequence numbers 1000 and 1002 followed by 999, it must discard 999.

If the subscriber builds on the AMPS Client Library, then nothing special is required to implement this logic. The following implementations allow you to track the latest message seen per publisher and to discard messages that have already been sent by a previous AMPS instance:

- `LoggedBookmarkStore`: used when you invoke `HAClient.createFileBacked()`
- `MemoryBookmarkStore`: used when you invoke `HAClient.createMemoryBacked()`

5. Conclusion

In this white paper, we looked at the issues that AMPS clients face in implementing high availability. If you choose to implement your own client for AMPS, these considerations will be of utmost importance to you as you decide exactly what level of reliability to target. If you build your application with the AMPS Client Library, you can greatly reduce complications by using HAClient along with the supplied `PublishStore` and `BookmarkStore` implementations. No matter what, understanding the choices and tradeoffs involved in implementing high availability will certainly help you to build the most robust and highestperformance solution.

FOR MORE INFORMATION, CONTACT:

60East Technologies

www.crankuptheamps.com [http://www.crankuptheamps.com]

`support@crankuptheamps.com`

(888) 206-1365