Platform SDK Developer's Guide

Migration from Protocol Manager Application Block Usage

4/20/2025

Contents

Migration from Protocol Manager Application Block Usage

Introduction

Starting with release 8.5.0, use of the Protocol Manager Application Block is no longer recommended. This application block is now considered a legacy component and has been deprecated.

This article provides an overview of how to migrate existing applications, and outlines how behavior that was previously handled by the application block should now be implemented.

Functional Aspects of the Protocol Manager Application Block

Protocols configurations helper classes:

- Protocols handshake options
- Connection configuration related options
- `WarmStandby` related options

Protocol Management Service functionality:

- `MessageReceiver` sharing feature
- `ChannelListener`'s sharing feature
- Protocols `WarmStandby` initialization feature
- Bulk `BeginOpen/BeginClose` functions

Functional Replacements

The usual Protocol Manager Application Block usage scenario is to help with the initialization of multiple protocol connections. Protocol Management configuration classes contain following parts:

- protocol handshake options,
- typified connection configuration options,
- `WarmStandby` options.

Connection Configuration Feature

[+] Java Code Sample

```
ProtocolManagementServiceImpl pmService = new ProtocolManagementServiceImpl();

TServerConfiguration config = new TServerConfiguration("t-server");

config.setUri(host, port); // - Target server host/port

config.setUseAddp(true); // - ConnectionConfiguration typified options like "UseAddp",
"AddpClientTimeout", etc
config.setAddpServerTimeout(addpServerTimeout);
config.setAddpClientTimeout(addpClientTimeout);

config.setFaultTolerance(FaultToleranceMode.WarmStandby);
config.setWarmStandbyUri(hostBackup, portBackup); // - Backup server host/port
config.setWarmStandbyAttempts((short) 3); // - WarmStandby typified options like
"WarmStandbyAttempts", etc
config.setWarmStandbyTimeout(2000);

config.setClientName(clientName); // - Protocol handshake typified options like "ClientName",
etc

Protocol protocol = pmService.register(config);
pmService.beginOpen();
```

[+] .NET Code Sample

```
var pmService = new ProtocolManagementService();
var config = new TServerConfiguration("t-server")
{
   Uri = new Uri("tcp://host:port/"), // - Target server host/port
   UseAddp = true,                    // - ConnectionConfiguration typified options like
"UseAddp", "AddpClientTimeout", etc
   AddpServerTimeout = addpServerTimeout,
   AddpClientTimeout = addpClientTimeout,
   FaultTolerance = FaultToleranceMode.WarmStandby,
   WarmStandbyUri = new Uri("tcp://backupHost:backupPort/"), // - Backup server host/port
   WarmStandbyAttempts = 3,                                  // - WarmStandby typified
options like "WarmStandbyAttempts", etc
   WarmStandbyTimeout = 2000,
   ClientName = clientName        // - Protocol handshake typified options like "ClientName",
etc
 };

IProtocol protocol = pmService.Register(config);
pmService.BeginOpen();
```

Last PSDK versions contain extended ConnectionConfiguration interfaces with set of typified properties for connections configuration. So, generally speaking, now we can initialize all of these options right with protocol connection initialization and it is not needed to have additional intermediate (duplicating) configuration structures.

Elimination of Protocol Management Service may look like:

[+] Java Code Sample

```
PropertyConfiguration connConf = new PropertyConfiguration();
```

```
connConf.setUseAddp(true); // - ConnectionConfiguration typified options like "UseAddp",
"AddpClientTimeout", etc
connConf.setAddpServerTimeout(addpServerTimeout);
connConf.setAddpClientTimeout(addpClientTimeout);


Endpoint endpoint = new Endpoint(epName1, host, port, connConf); // - Target server host/port
are here
Endpoint endpointBackup = new Endpoint(epName2, hostBackup, portBackup, connConf); // -
Backup server host/port are here

TServerProtocol protocol = new TServerProtocol(endpoint);
protocol.setClientName(clientName); // - Protocol handshake typified options like
"ClientName", etc

WarmStandbyConfiguration wsConf = new WarmStandbyConfiguration(endpoint, endpointBackup);
wsConf.setAttempts((short) 3); // - WarmStandby typified options like "WarmStandbyAttempts",
etc
wsConf.setTimeout(2000);
WarmStandbyService wsService = new WarmStandbyService(protocol);
wsService.applyConfiguration(wsConfig);
wsService.start();
protocol.beginOpen();
```

[+] .NET Code Sample

```
// - ConnectionConfiguration typified options like "UseAddp", "AddpClientTimeout", etc
PropertyConfiguration connConf = new PropertyConfiguration()
{
   UseAddp = true,
   AddpServerTimeout = addpServerTimeout,
   AddpClientTimeout = addpClientTimeout
};

var endpoint = new Endpoint(epName1, host, port, connConf); // - Target server host/port are
here
var endpointBackup = new Endpoint(epName2, hostBackup, portBackup, connConf); // - Backup
server host/port are here

var protocol = new TServerProtocol(endpoint);
protocol.ClientName = clientName; // - Protocol handshake typified options like "ClientName",
etc
var wsConf = new WarmStandbyConfiguration(endpoint, endpointBackup);
wsConf.Attempts = 3; // - WarmStandby typified options like "WarmStandbyAttempts", etc
wsConf.Timeout = 2000;

WarmStandbyService wsService = new WarmStandbyService(protocol);
wsService.ApplyConfiguration(wsConf);
wsService.Start();
protocol.BeginOpen();
```

MessageReceiver Sharing Feature

The Protocol Manager Service instance unconditionally uses its own instance of `MessageReceiver` for all of its registered protocols.

So, it is not possible to use `protocol.receive()` on protocols instances created with `ProtocolManagementService`. It may be done with `pmServiceImpl.getReceiver().receive()`. This method returns asynchronous incoming message from shared queue of all protocols registered with

this `pmServiceImpl`.

Actually, `MessageReceiver` mechanism is deprecated and, usually, it is not effective to use single handler/queue to process event messages from different protocols connections.

Application logic will be more clear and supportable when each protocol has own specific event handling logic.

So, the recommendation is to use specific `MessageHandler` for protocol instance where it is required. For example:

[+] Java Code Sample

```
protocol.setMessageHandler(new MessageHandler() {
        public void onMessage(final Message message) {
            // do fast event message procession or pass it to some other executor for handling
        }
});
```

[+] .NET Code Sample

```
protocol.Received += (sender, args) =>
{
   IMessage message = ((MessageEventArgs) args).Message;
   // do fast event message procession or pass it to some other executor for handling
};
```

ChannelListener Events Concentration

Protocol Management service supports notifications to set of clients `ChannelListeners`.

To migrate out of Protocol Manager Application Block usage, `ChannelListeners` may be added directly to protocol connections.

By the way, in most cases situation like "publish channel events from all of the N connections to all of the M listeners" is a kind of an application design issue, though, it is possible to be realized.

WarmStandby Service Initialization

Protocol Manager service instance does initialize `WarmStandby` service internally if given protocol configuration contains `WarmStandby` related options/values.

[+] Java Code Sample

```
...
config.setFaultTolerance(FaultToleranceMode.WarmStandby);

config.setWarmStandbyUri(hostBackup, portBackup);
config.setWarmStandbyAttempts((short) 3);
config.setWarmStandbyTimeout(2000);
...
```

[+] .NET Code Sample

```
var config = new ConfServerConfiguration("confserver")
{
   ...
   FaultTolerance = FaultToleranceMode.WarmStandby,
   WarmStandbyUri = new Uri("tcp://backupHost:backupPort/"),
   WarmStandbyAttempts = 3,
   WarmStandbyTimeout = 2000,
   ...
 };
```

The recommendation is to create and initialize it explicitly. Initialization schema was mentioned above. Initialization may look like:

[+] Java Code Sample

```
Endpoint confEPprimary = ...;
Endpoint confEPbackup = ...;

ConfServerProtocol cfgProtocol = new ConfServerProtocol(confEPprimary);
cfgProtocol.setClientName(appName);
cfgProtocol.setClientApplicationType(appType);
cfgProtocol.setUserName(username);
cfgProtocol.setUserPassword(password);

WarmStandbyConfiguration wsConfig = new WarmStandbyConfiguration(confEPprimary, confEPbackup);
wsConfig.setTimeout(2000);
wsConfig.setAttempts((short) 3);
WarmStandbyService wsService = new WarmStandbyService(cfgProtocol);
wsService.applyConfiguration(wsConfig);
wsService.start();

cfgProtocol.beginOpen();
```

[+] .NET Code Sample

```
Endpoint confEPprimary = ...;
Endpoint confEPbackup = ...;

ConfServerProtocol cfgProtocol = new ConfServerProtocol(confEPprimary);
cfgProtocol.ClientName = appName;
cfgProtocol.ClientApplicationType = appType;
cfgProtocol.UserName = username;
cfgProtocol.UserPassword = password;

WarmStandbyConfiguration wsConfig = new WarmStandbyConfiguration(confEPprimary, confEPbackup);
wsConfig.Timeout = 2000;
wsConfig.Attempts = 3;
WarmStandbyService wsService = new WarmStandbyService(cfgProtocol);
wsService.ApplyConfiguration(wsConfig);
wsService.Start();

cfgProtocol.BeginOpen();
```

For more details, see WarmStandby Application Block documentation.

Bulk Open/Close Functions

Actually, `pmService.beginOpen()` means `protocol.beginOpen()` for all protocols registered in given

PM Service.