Application Development Guide

Platform Symphony™
Version 4.1
November 2008

Copyright

We’d like to hear from you

You can help us make this document better by telling us what you think of the content, organization, and usefulness of the information. If you find an error, or just want to make a suggestion for improving this document, please address your comments to doc@platform.com.

Your comments should pertain only to Platform documentation. For product support, contact support@platform.com.

Document redistribution and translation

Internal redistribution

Trademarks

Third-party license agreements

http://www.platform.com/Company/third.part.license.htm

Third-party copyright notices

http://www.platform.com/Company/Third.Party.Copyright.htm

Contents

PART

Application Development

CHAPTER

1

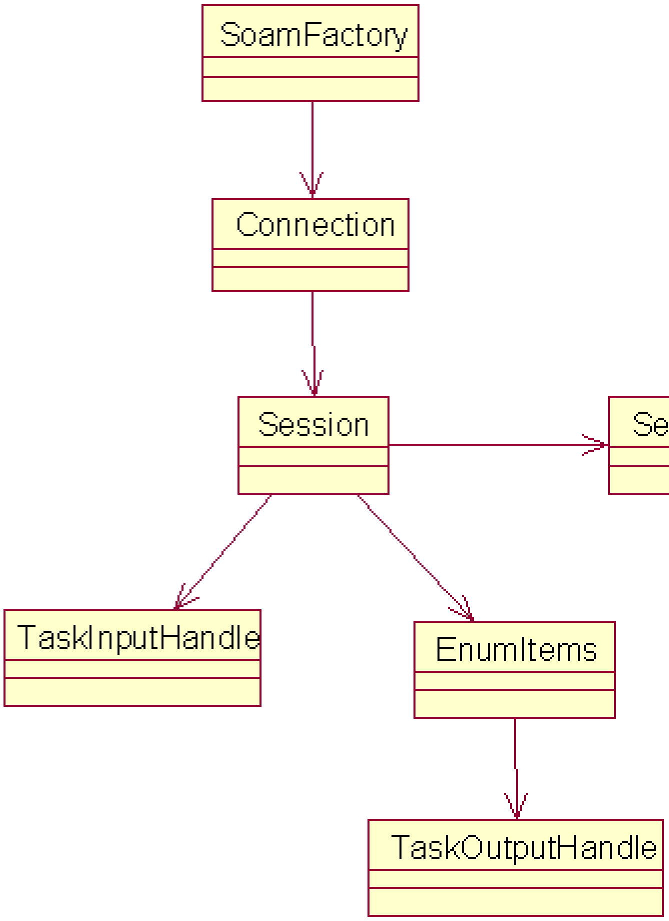
Overview of API Classes

Symphony API classes

Client classes

Workload management

The following diagram describes the relationships of the API classes that manage session workload.



SoamFactory

Initializes the Symphony API

Connection

Used by the client of an application to maintain a physical connection between the client and Symphony

Session

Enables the client to manage its workload

SessionCallback

Called when one task response is ready

TaskInputHandle

Gets the ID of a task Symphony returned

EnumItems

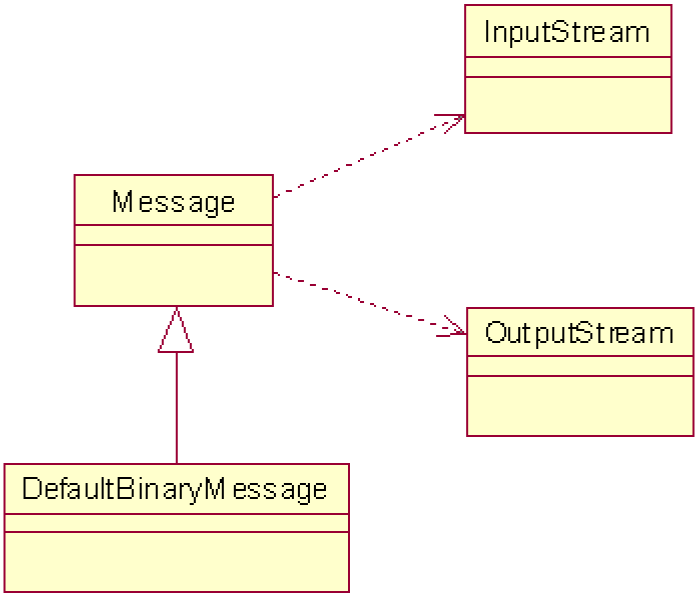
Returns the next item in the enumeration

TaskOutputHandle

Hosts a task ID, a message response (if any), and an exception (if any) from Symphony

Input and output

The following diagram describes the relationships between the API classes that manage communications between Symphony and the client.



DefaultBinary Message

A default implementation for a binary data message

Message

Used to deliver messages between Symphony and the client

InputStream

A stream object that the message object uses for reading

OutputStream

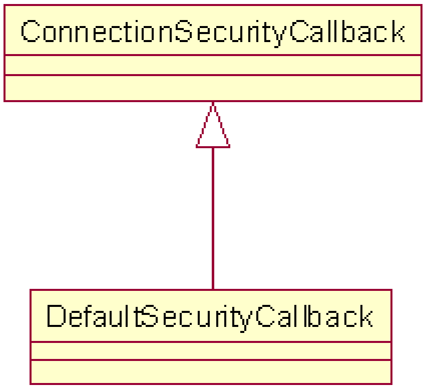
A stream object that is used by a message object for writing

In the diagrams that follow, dotted lines indicate related classes. Solid lines indicate inheritance.

Common classes

Security

To ensure system security, the following API classes must be implemented.



DefaultSecurity Callback

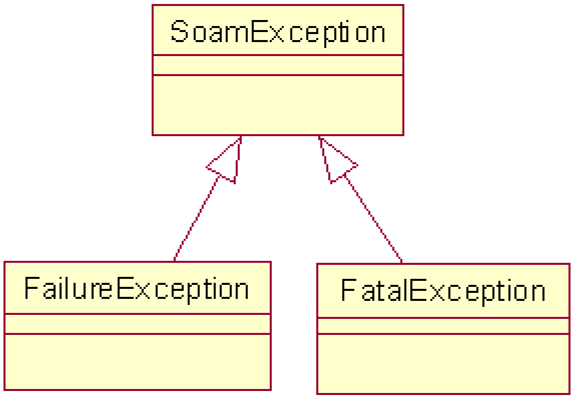
The default security implementation invoked when a new Symphony connection requires a security token

ConnectionSecurity Callback

Invoked when making a new connection to Symphony

Exceptions

Client applications and services can throw two types of exceptions, `FailureException` and `FatalException`. This diagram shows how they relate to the base exception class, `SoamException`.



SoamException

Base class of exceptions the system generates. Can be accessed by client applications and services

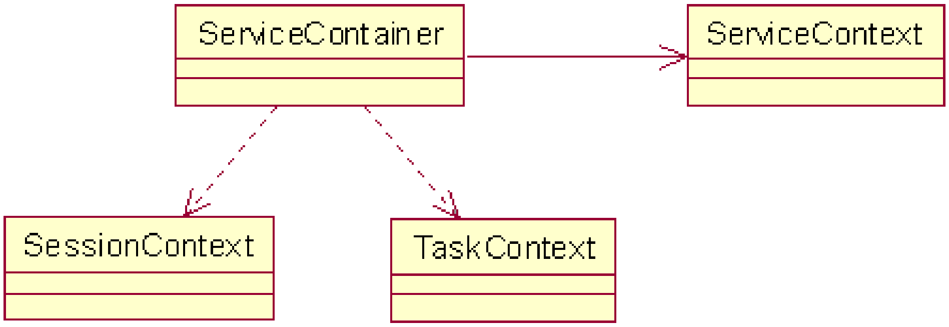
FailureException

Exception thrown to indicate a non-fatal error has occurred in a service. The action taken in the event of a FailureException is dependent on the API method that the exception is thrown in.

FatalException

Exception thrown to indicate a fatal error has occurred in a service. The action taken in the event of a FatalException is dependent on the API method that the exception is thrown in.

Service classes



ServiceContainer

Implements all the required methods to allow the system to interact with the service instance

ServiceContext

Provides functionality that the service requires throughout its lifetime

SessionContext

Hosts information that may be required while servicing a task from a session

TaskContext

Provides functionality that a service invocation requires

CHAPTER

2

Getting Started: SampleApp

Tutorial: Synchronous Symphony C++ client tutorial

Goal

This tutorial guides you through the process of building, packaging, deploying, and running the hello grid sample client and service. It also walks you through the sample client application code.

You learn the minimum amount of code that you need to create a client.

At a glance

Before you begin, ensure you have installed and started Platform Symphony DE. You complete the following tasks:

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

You can build client application and service samples at the same time.

1. Locate workspace file `sampleCPP_vc6.dsw`, solution file `sampleCPP_vc71.sln`, or solution file `sampleCPP_vc80.sln` in `%SOAM_HOME%\4.1\samples\CPP\SampleApp`.
2. Load the file into Visual Studio and build it.

On Linux

You can build client application and service samples at the same time.

1. Change to the `conf` directory under the directory in which you installed Symphony DE.

   For example, if you installed Symphony DE in `/opt/symphonyDE/DE41`, go to `/opt/symphonyDE/DE41/conf`.
2. Source the environment:
   - For `csh`, enter

     ```
     source cshrc.soam
     ```
   - For `bash`, enter

     ```
     . profile.soam
     ```
3. Compile using the Makefile located in `$SOAM_HOME/4.1/samples/CPP/SampleApp`:

   ```
   make
   ```

Package the sample service

On Windows

To deploy the service, you first need to package it.

1. Go to the directory in which the compiled samples are located.

   ```
   cd %SOAM_HOME%\4.1\samples\CPP\SampleApp\Output\
   ```

2. Create the service package by compressing the service executable into a zip file.

   ```
   gzip SampleServiceCPP.exe
   ```

   You have now created your service package `SampleServiceCPP.exe.gz`.

On Linux

To deploy the service, you first need to package it.

1. Change to the directory in which the compiled samples are located:

   ```
   cd $SOAM_HOME/4.1/samples/CPP/SampleApp/Output/
   ```

2. Create the service package by archiving and compressing the service executable file:

   ```
   tar -cvf SampleServiceCPP.tar SampleServiceCPP
   ```

   ```
   gzip SampleServiceCPP.tar
   ```

   You have now created your service package `SampleServiceCPP.tar.gz`.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.

2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.

3. Select Add an application, then click Continue.

   The Adding an Application page displays.

4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue.

   For SampleApp, you can find your profile in the following location:

   - Windows—`%SOAM_HOME%\4.1\samples\CPP\SampleApp\SampleApp.xml`
   - Linux—`$SOAM_HOME/4.1/samples/CPP/SampleApp/SampleApp.xml`

   The Service Package location window displays.

6. Browse to the service package you created in .gz or tar.gz format and select it. Click Continue.

   The Confirmation window displays.

7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.

8. Click Close.

The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   %SOAM_HOME%\4.1\samples\CPP\SampleApp\Output\SyncClient.exe

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

1. Run the client application:

   $SOAM_HOME/4.1/samples/CPP/SampleApp/Output/SyncClient

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Review and understand the samples

You review the sample client application code to learn how you can create a synchronous client application.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\SyncClient`
	Message object	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Common`
	Service code	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\4.1\samples\CPP\SampleApp\SampleApp.xml`
	Output directory	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Output\`

Operating System	Files	Location of Code Sample
Linux	Client	`$SOAM_HOME/4.1/samples/CPP/SampleApp/SyncClient`
	Message object	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Common`
	Service code	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/4.1/samples/CPP/SampleApp/SampleApp.xml`
	Output directory	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Output/`

What the sample does

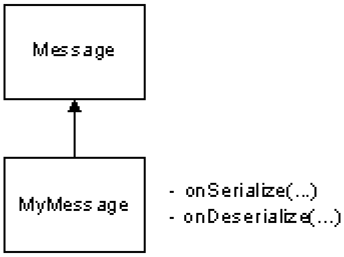
The client application sample opens a session and sends 10 input messages, and retrieves the results. The client application is a synchronous client that sends input and blocks the output until all the results are returned.

The service takes input data sent by client applications, returns the input data you have sent and replies "Hello Client!!"

Review the sample code

Input and output: declare the message object

Your client application needs to handle data that it sends as input, and output data that it receives from the service.



Tip:

Client applications and services share the same message class.

In `MyMessage.h`:

- We declare the `MyMessage` class
- We define serialization methods for input and output messages
- We implement methods to handle the data

Note:

For this example, we have defined the same class for input and output messages. However, you can define separate classes for input and output messages.

```
#pragma once
#include "soam.h"
class MyMessage :
    public soam::Message
{
public:
    MyMessage();
    MyMessage(int i, bool isSync, char* str);
    virtual ~MyMessage(void);
    void onSerialize(
        /*[in]*/ soam::OutputStreamPtr &stream) throw (soam::SoamException);
    void onDeserialize(
    /*[in]*/ soam::InputStreamPtr &stream) throw (soam::SoamException);
// accessors
public:
    int getInt() const{return m_int;}
    void setInt(int _int) {m_int = _int;}
    const char* getString() {return m_string;}
    void setString(const char* str) {freeString(m_string); m_string = copyString(str);}
    bool getIsSync() const{return (m_isSync != 0);}
    void setIsSync(bool isSync) {m_isSync = isSync;}
private:
    char* copyString(const char* strSource);
    void freeString(char* strToFree);
private:
    int m_int;
    bool m_isSync;
    char* m_string;
};
```

Implement the MyMessage object

Once your message object is declared, implement handlers for serialization and deserialization.

In `MyMessage.cpp`, we implement methods to handle the data. For data types that are supported by Symphony DE, see the appropriate API reference.

Note:

If you already have an application with a message object that is serialized, you can pass a binary blob through the DefaultBinaryMessage class.

```
#include "stdafx.h"
#include <string.h>
#include "MyMessage.h"
#include "soam.h"
using namespace soam;
MyMessage::MyMessage()
{
    m_int = 0;
    m_string = copyString("");
}
MyMessage::MyMessage(int i, bool isSync, char* str)
{
    m_int = i;
    m_isSync = isSync;
    m_string = copyString(str);
}
MyMessage::~MyMessage(void)
{
    freeString(m_string);
}void MyMessage::onSerialize(OutputStreamPtr &stream) throw (SoamException)
{
    stream->write(m_int);
    stream->write(m_isSync);
    stream->write(m_string);
}void MyMessage::onDeserialize(InputStreamPtr &stream) throw (SoamException)
{
    stream->read(m_int);
    stream->read(m_isSync);
    freeString(m_string);
    stream->read(m_string);
}char* MyMessage::copyString(const char* strSource)
{
    SOAM_ASSERT(0 != strSource);
    size_t len = strlen(strSource);
    char* newString = new char[len+1];
    SOAM_ASSERT(0 != newString);
    strcpy(newString, strSource);
    return newString;
}
void MyMessage::freeString(char* strToFree)
{
    if (0 != strToFree)
    {
        delete []strToFree;
    }
}
```

Initialize the client

In `SyncClient.cpp`, when you initialize, you initialize the Symphony client infrastructure. You initialize once per client.

Important:

Initialization is required. Otherwise, API calls fail.

```
...
    try
    {
        // Initialize the API
        SoamFactory::initialize();
...
```

Connect to an application

To send data to be calculated in the form of input messages, you connect to an application.

You specify an application name, a user name, and password. The application name must match that defined in the application profile.

For Symphony DE, there is no security checking and login credentials are ignored—you can specify any user name and password. Security checking is done however, when your client application submits workload to the actual grid.

The default security callback encapsulates the callback for the user name and password.

Tip:

When you connect, a connection object is returned.

```
...
  // Set up application specific information to be supplied to Symphony
     char appName[]="SampleAppCPP";
  // Set up application authentication information using the default security provider
     DefaultSecurityCallback securityCB("Guest", "Guest");
  // Connect to the specified application
     ConnectionPtr conPtr = SoamFactory::connect(appName, &securityCB);
  // Retrieve and print our connection ID
     cout << "connection ID=" << conPtr->getId() << endl;
...
```

Create a session to group tasks

A session is a way of logically grouping tasks that are sent to a service for execution. The tasks are sent and received synchronously.

When creating a session, you need to specify the session attributes by using the SessionCreationAttributes object. In this sample, we create a SessionCreationAttributes object called attributes and set three parameters in the object.

The first parameter is the session name. This is optional. The session name can be any descriptive name you want to assign to your session. It is for information purposes, such as in the command-line interface.

The second parameter is the session type. The session type is optional. If you leave this parameter blank " " or do not set a session type, system default values are used for session attributes. If you specify a session type in the client application, you must also configure the session type in the application profile—the session type name in your application profile and session type you specify in the client must match. If you use an incorrect session type in the client and the specified session type cannot be found in the applicatin profile, an exception is thrown to the client.

The third parameter is the session flag, which we specify as ReceiveSync. You must specify it as shown. This indicates to Symphony that this is a synchronous session.

We pass the attributes object to the createSession() method, which returns a pointer to the session.

```
        // Set up session creation attributes
        SessionCreationAttributes attributes;
        attributes.setSessionName("mySession");
        attributes.setSessionType("ShortRunningTasks");
        attributes.setSessionFlags(Session::ReceiveSync);
        // Create a synchronous session
        SessionPtr sesPtr = conPtr->createSession(attributes);
```

Send input data to be processed

In this step, we create 10 input messages to be processed by the service. When a message is sent, a task input handle is returned. This task input handle contains the ID for the task that was created for this input message.

```
 int tasksToSend = 10;
        for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
        {
            // Create a message
            char hello[]="Hello Grid !!";
            MyMessage inMsg(taskCount, true, hello);
            // Create task attributes
            TaskSubmissionAttributes attrTask;
            attrTask.setTaskInput(&inMsg);
                        // send it
            TaskInputHandlePtr input = sesPtr->sendTaskInput(attrTask);
            // Retrieve and print task ID
            cout << "task submitted with ID : " << input->getId() << endl;
        }
...
```

Retrieve output

Pass the number of tasks to the fetchTaskOutput() method to retrieve the output messages that were produced by the service. This method blocks until the output for all tasks is retrieved. The return value is an enumeration that contains the completed task results. Iterate through the task results and extract the messages using the populateTaskOutput() method. Display the task ID and the results from the output message.

```
        // Now get our results - will block here until all tasks retrieved
        EnumItemsPtr enumOutput = sesPtr->fetchTaskOutput(tasksToSend);
        // Inspect results
        TaskOutputHandlePtr output;
        while(enumOutput->getNext(output))
        {
            // Check for success of task
            if (true == output->isSuccessful())
            {
                // Get the message returned from the service
                MyMessage outMsg;
                output->populateTaskOutput(&outMsg);
                // Display content of reply
                cout << "Task Succeeded [" <<  output->getId() << "]" << endl;
                cout << outMsg.getResult() << endl;
            }
            else
            {
                // Get the exception associated with this task
                SoamExceptionPtr ex = output->getException();
                cout << "Task Failed : " << ex->what() << endl;
            }
        }
```

Catch exceptions

Any exceptions thrown take the form of SoamException. Catch all Symphony exceptions to know about exceptions that occurred in the client application, service, and middleware.

The sample code above catches exceptions of type SoamException.

```
catch(SoamException& exp)
{
// Report exception
cout << "exception caught ... " << exp.what() << endl;
}
```

Uninitialize

Always uninitialize the client API at the end of all API calls. If you do not call uninitialize, the client API is in an undefined state, resources used by the client are held indefinitely, and there is no guarantee your client will be stable.

Important:

Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

```
    // uninitialize the API
    // This is the only means to ensure proper shutdown
    // of the interaction between the client and the system.
    SoamFactory::uninitialize();
...
```

Tutorial: SampleApp: Developing an asynchronous Symphony C++ client

Goal

The purpose of an asynchronous client is to get the output as soon as it is available. The client thread does not need to be blocked once the input data is sent and can perform other actions.

In this tutorial, you learn how to convert a synchronous client into asynchronous.

At a glance

Before you begin, ensure you have installed and started Platform Symphony DE. You complete the following tasks:

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

You can build client application and service samples at the same time.

1. Locate workspace file `sampleCPP_vc6.dsw`, solution file `sampleCPP_vc71.sln`, or solution file `sampleCPP_vc80.sln` in `%SOAM_HOME%\4.1\samples\CPP\SampleApp`.
2. Load the file into Visual Studio and build it.

On Linux

You can build client application and service samples at the same time.

1. Change to the `conf` directory under the directory in which you installed Symphony DE.

   For example, if you installed Symphony DE in `/opt/symphonyDE/DE41`, go to `/opt/symphonyDE/DE41/conf`.
2. Source the environment:
   - For `csh`, enter

     ```
     source cshrc.soam
     ```
   - For `bash`, enter

     ```
     . profile.soam
     ```
3. Compile using the Makefile located in `$SOAM_HOME/4.1/samples/CPP/SampleApp`:

   make

Package the sample service

On Windows

To deploy the service, you first need to package it.

1. Go to the directory in which the compiled samples are located.

   ```
   cd %SOAM_HOME%\4.1\samples\CPP\SampleApp\Output\
   ```
2. Create the service package by compressing the service executable into a zip file.

   ```
   gzip SampleServiceCPP.exe
   ```

   You have now created your service package `SampleServiceCPP.exe.gz`.

On Linux

To deploy the service, you first need to package it.

1. Change to the directory in which the compiled samples are located:

   ```
   cd $SOAM_HOME/4.1/samples/CPP/SampleApp/Output/
   ```
2. Create the service package by compressing the service executable into a tar file:

   ```
   tar -cvf SampleServiceCPP.tar SampleServiceCPP
   ```

   ```
   gzip SampleServiceCPP.tar
   ```

   You have now created your service package `SampleServiceCPP.tar.gz`.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue.

   For SampleApp, you can find your profile in the following location:

   - Windows—`%SOAM_HOME%\4.1\samples\CPP\SampleApp\SampleApp.xml`
   - Linux—`$SOAM_HOME/4.1/samples/CPP/SampleApp/SampleApp.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .gz or tar.gz format and select it, then, click Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   %SOAM_HOME%\4.1\samples\CPP\SampleApp\Output\AsyncClient.exe

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

1. Run the client application:

   $SOAM_HOME/4.1/samples/CPP/SampleApp/Output/AsyncClient

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

You review the sample client application code to learn how you can understand the differences between a synchronous client and an asynchronous client.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\AsyncClient`
	Message object	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Common`
	Service code	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\4.1\samples\CPP\SampleApp\SampleApp.xml`
	Output directory	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Output\`

Operating System	Files	Location of Code Sample
Linux	Client	`$SOAM_HOME/4.1/samples/CPP/SampleApp/AsyncClient`
	Message object	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Common`
	Service code	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/4.1/samples/CPP/SampleApp/SampleApp.xml`
	Output directory	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Output/`

What the sample does

The client application sample sends 10 input messages with the data Hello Grid!! and retrieves the results.

Results are returned asynchronously with a callback interface provided by the client to the API. Methods on this interface are called from threads within the API when certain events occur. In the sample, the events are:

- When there is an error at the session level
- When results return from Symphony

Considerations for asynchronous clients

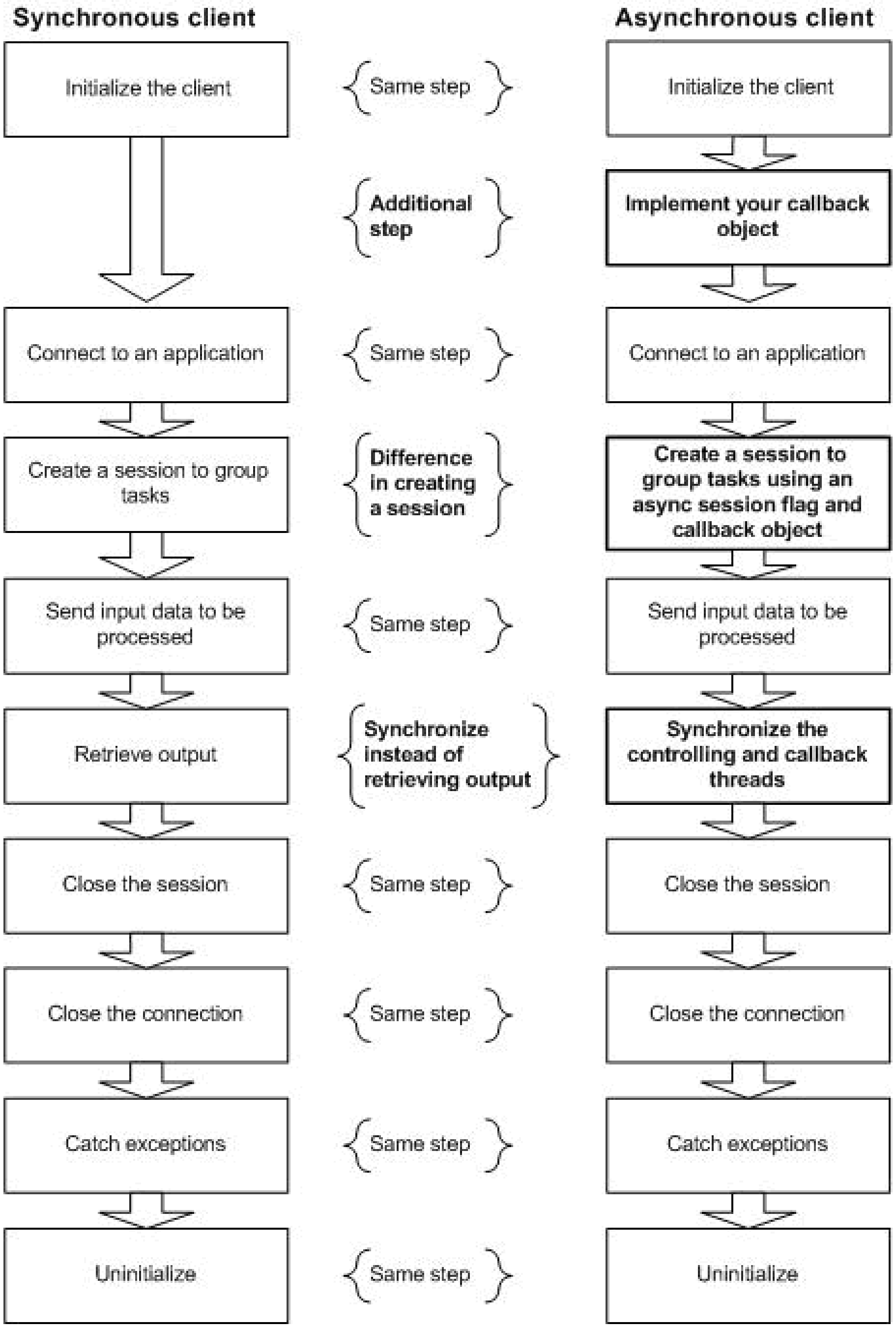
Synchronization	Because results can come back at any time, it is probable that your callback code needs synchronization between the callback thread and the controlling thread. The controlling thread needs to know when work is complete.
Order of results	Results are not sent back in order. If order of results is important, the client application must sort the results.

Code differences between synchronous and asynchronous clients

An asynchronous client is very similar to a synchronous client. The only differences are:

- You need to specify a callback when creating a session
- You specify a different flag to indicate asynchronous when you create a session
- Retrieval of replies

Let us look at the steps to create synchronous and asynchronous clients and highlight the differences. Steps in bold indicate differences. Everything else is the same as the synchronous client.


Synchronous client
Asynchronous client
Initialize the client
Same step
Initialize the client
Additional step
Implement your callback object
Connect to an application
Same step
Connect to an application
Create a session to group tasks
Difference in creating a session
Create a session to group tasks using an async session flag and callback object
Send input data to be processed
Same step
Send input data to be processed
Retrieve output
Synchronize instead of retrieving output
Synchronize the controlling and callback threads
Close the session
Same step
Close the session
Close the connection
Same step
Close the connection
Catch exceptions
Same step
Catch exceptions
Uninitialize
Same step
Uninitialize

Declare the message object and implement

As in the synchronous client tutorial, declare the message object and implement your own message object.

If you have not done so already, take a look at the synchronous client application tutorial Your First Synchronous Symphony C++ Client for details on the Message object, specifically:

- Input and output: declare the message object

- Implement the MyMessage object

Declare and implement your callback object

Perform this step after declaring the Message object and implementing the MyMessage object.

In `MyCallback.h`, we create our own callback class from the SessionCallback class, and we implemented onResponse() to retrieve the output for each input message that we send.

Note:

- onResponse() is called every time a task completes and output is returned to the client. The task output handle allows the client code to manipulate the output.
- isSuccessful() checks whether there is output to retrieve.
- If there is output to retrieve, populateTaskOutput() gets the output. Once results return, we print them to standard output and return.

```
#include "soam.h"
using namespace soam;
using namespace std;
#ifndef WIN32
#include <pthread.h>
#else
#include <windows.h>
#endif
class MySessionCallback :
    public SessionCallback
{
    public:
        MySessionCallback()
            :m_tasksReceived(0), m_exception(false)
        {
#ifndef WIN32
            pthread_mutexattr_t attr;
            pthread_mutexattr_init( &attr );
            pthread_mutexattr_settype( &attr, PTHREAD_MUTEX_RECURSIVE );
            pthread_mutex_init( &m_mutex, &attr );
            pthread_mutexattr_destroy( &attr );
#else
            InitializeCriticalSection(&m_criticalSection);
#endif
            cout << "Callback created ... " << endl;
        }
virtual ~MySessionCallback()
        {
#ifndef WIN32
            pthread_mutex_destroy( &m_mutex );
#else
            DeleteCriticalSection(&m_criticalSection);
#endif
        }
```

```
/////////////////////////////////////////////////////////
// This handler is called once any exception occurs
// within the scope of the session.
// ======================================================
virtual void onException(SoamException &exception) throw()
{
cout << "An exception occured on the callback.\nDetails : " <<  exception.what() << endl;
#ifndef WIN32
                pthread_mutex_lock( &m_mutex);
#else
                EnterCriticalSection(&m_criticalSection);
#endif
                m_exception = true;
#ifndef WIN32
                pthread_mutex_unlock( &m_mutex);
#else
                LeaveCriticalSection(&m_criticalSection);
#endif
}
/////////////////////////////////////////////////////////
// This handler is called once a message is returned
// from the system when a task completes.
// ======================================================
void onResponse(TaskOutputHandlePtr &output) throw()
        {
            try
            {
                // check for success of task
                if (true == output->isSuccessful())
                {                    // get the message returned from the service
                    MyMessage outMsg;
                    output->populateTaskOutput(&outMsg);
                    // display content of reply
                    cout << "Task Succeeded [" <<  output->getId() << "]" << endl;
                    cout << "Integer Value : " << outMsg.getInt() << endl;
                    cout << outMsg.getString() << endl;
                }
                else
                {
                    // get the exception associated with this task
                    SoamExceptionPtr ex = output->getException();
                    cout << "Task Failed : " << ex->what() << endl;
                }
            }
            catch(SoamException &exception)
            {
                cout << "Exception occured in OnResponse() : " << exception.what() << endl;
            }
// Update counter used to synchronize the controlling thread
// with this callback object
#ifndef WIN32
                pthread_mutex_lock( &m_mutex);
#else
                EnterCriticalSection(&m_criticalSection);
#endif
                ++m_tasksReceived;
#ifndef WIN32
                pthread_mutex_unlock( &m_mutex);
#else
                LeaveCriticalSection(&m_criticalSection);
#endif
        }

        inline long getReceived()
        {
            return m_tasksReceived;
        }
        inline bool getDone()
        {
            return m_exception;
        }

private:
#ifndef WIN32
        pthread_mutex_t m_mutex;
#else
        CRITICAL_SECTION m_criticalSection;
#endif
        long m_tasksReceived;
        bool m_exception;
};
```

Create a session to group tasks

In `AsyncClient.cpp`, perform this step after you have connected to the application.

When creating an asynchronous session, you need to specify the session attributes by using the SessionCreationAttributes object. In this sample, we create a SessionCreationAttributes object called attributes and set four parameters in the object.

The first parameter is the session name. This is optional. The session name can be any descriptive name you want to assign to your session. It is for information purposes, such as in the command line interface.

The second parameter is the session type. The session type is optional. You can leave this parameter blank or not make the API call at all. When you do this, system default values are used for your session.

The third parameter is the session flag, which we specify as ReceiveAsync. You must specify it as shown. This indicates to Symphony that this is an asynchronous session.

The fourth parameter is the callback object.

We pass the attributes object to the createSession() method, which returns a pointer to the session.

```
...
        // Create session callback
        MySessionCallback  myCallback;
        // Set up session creation attributes
        SessionCreationAttributes attributes;
        attributes.setSessionName("mySession");
        attributes.setSessionType("ShortRunningTasks");
        attributes.setSessionFlags(Session::ReceiveAsync);
        attributes.setSessionCallback(&myCallback);
        // Create an asynchronous session
        SessionPtr sesPtr = conPtr->createSession(attributes);
...
```

Synchronize the controlling and callback threads

Perform this step after sending the input data to be processed.

Since our client is asynchronous, we need to synchronize the controlling thread and the callback thread. In this example, the controlling thread blocks until all replies have come back.

```
...
// Now wait until all replies have been received asynchronously
// by our callback ... for illustrative purposes we will poll
// until all replies are in.
while ((myCallback.getReceived() < tasksToSend) && !myCallback.getDone())
 {
   ourSleep(2);
 }
...
```

Tutorial: SampleApp: Your first Symphony C++ service

Goal

This tutorial guides you through the process of building and running a service, then walks you through the sample service code.

You learn the minimum amount of code that you need to create a service.

At a glance

Before you begin, ensure you have installed and started Platform Symphony DE.

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

You can build client application and service samples at the same time.

1. Locate workspace file `sampleCPP_vc6.dsw`, solution file `sampleCPP_vc71.sln`, or solution file `sampleCPP_vc80.sln` in `%SOAM_HOME%\4.1\samples\CPP\SampleApp`.
2. Load the file into Visual Studio and build it.

On Linux

You can build client application and service samples at the same time.

1. Change to the `conf` directory under the directory in which you installed Symphony DE.

   For example, if you installed Symphony DE in `/opt/symphonyDE/DE41`, go to `/opt/symphonyDE/DE41/conf`.
2. Source the environment:
   - For `csh`, enter

     ```
     source cshrc.soam
     ```
   - For `bash`, enter

     ```
     . profile.soam
     ```
3. Compile using the Makefile located in `$SOAM_HOME/4.1/samples/CPP/SampleApp`:

   ```
   make
   ```

Package the sample service

On Windows

To deploy the service, you first need to package it.

1. Go to the directory in which the compiled samples are located.

   ```
   cd %SOAM_HOME%\4.1\samples\CPP\SampleApp\Output\
   ```
2. Create the service package by compressing the service executable into a zip file.

   ```
   gzip SampleServiceCPP.exe
   ```

   You have now created your service package `SampleServiceCPP.exe.gz`.

On Linux

To deploy the service, you first need to package it.

1. Change to the directory in which the compiled samples are located:

   ```
   cd $SOAM_HOME/4.1/samples/CPP/SampleApp/Output/
   ```
2. Create the service package by compressing the service executable into a tar file:

   ```
   tar -cvf SampleServiceCPP.tar SampleServiceCPP
   ```

   ```
   gzip SampleServiceCPP.tar
   ```

   You have now created your service package `SampleServiceCPP.tar.gz`.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue.

   For SampleApp, you can find your profile in the following location:

   - Windows—`%SOAM_HOME%\4.1\samples\CPP\SampleApp\SampleApp.xml`
   - Linux—`$SOAM_HOME/4.1/samples/CPP/SampleApp/SampleApp.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .gz or tar.gz format and select it, then, click Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

```
%SOAM_HOME%\4.1\samples\CPP\SampleApp\Output\SyncClient.exe
```

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

1. Run the client application:

```
$SOAM_HOME/4.1/samples/CPP/SampleApp/Output/SyncClient
```

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

You review the sample service code to learn how you can create a service.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\SyncClient`
	Message object	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Common`
	Service code	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\4.1\samples\CPP\SampleApp\SampleApp.xml`
	Output directory	`%SOAM_HOME%\4.1\samples\CPP\SampleApp\Output`

Operating System	Files	Location of Code Sample
Linux	Client	`$SOAM_HOME/4.1/samples/CPP/SampleApp/SyncClient`
	Message object	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Common`
	Service code	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/4.1/samples/CPP/SampleApp/SampleApp.xml`
	Output directory	`$SOAM_HOME/4.1/samples/CPP/SampleApp/Output/`

What the sample does

The service takes input data sent by client applications, returns the input data you have sent and replies "Hello Client !!"

Input and output: declare and implement the Message object:

Your service needs to handle data that it receives as input, and generate output data that can be sent back to the client application.

Note:

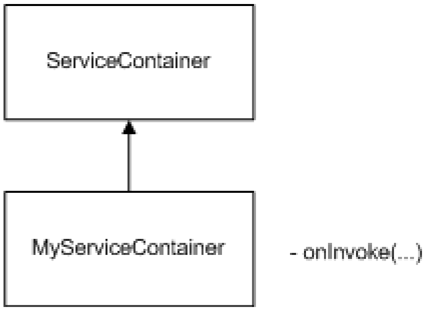
Client applications and services share the same message class. You do not need to create a different message class. In our example, we have created a common directory for code that is shared by client and service. Use the Message object declared and implemented by the client application.

If you have not done so already, take a look at the synchronous client application tutorial for details on the Message object.

- Input and output: declare the message object:
- Implement the MyMessage object:

Define a service container:

For a service to be managed by Symphony, it needs to be in a container object. This is the service container.



In `SampleService.cpp`, we inherited from the ServiceContainer class.

```
#include "stdafx.h"
#include <stdio.h>
#include "soam.h"
#include "MyMessage.h"
using namespace soam;
using namespace std;
class MyServiceContainer : public ServiceContainer
```

Process the input:

Symphony calls onInvoke() on the service container once per task. Once you inherit from the ServiceContainer class, implement handlers so that the service can function properly. This is where the calculation is performed.

To gain access to the data set for the client, you must present an instance of the message object to the populateTaskInput() method on the task context.

The task context contains all information and functionality that is available to the service during an onInvoke() call in relation to the task that is being processed.

Important:

Services are virtualized. As a result, a service should not read from stdin or write to stdout. Services can, however, read from and write to files that are accessible to all compute hosts.

You present to populateTaskInput() the message object that comes from the client application. During this call, the data sent from the client is used to populate the message object.

```
{
public:
    virtual void onInvoke (TaskContextPtr& taskContext)
    {
        // get the input that was sent from the client
        MyMessage inMsg;
        taskContext->populateTaskInput(inMsg);
        // We simply echo the data back to the client
        MyMessage outMsg;
        outMsg.setInt(inMsg.getInt());
        std::string str="you sent : ";
        str += inMsg.getString();
        str += "\nwe replied : Hello Client !!\n>>> ";
        if (inMsg.getIsSync())
        {
            str += "Synchronously.\n";
        }
        else
        {
            str += "Asynchronously.\n";
        }
        outMsg.setString(str.c_str());
        // set our output message
        taskContext->setTaskOutput(outMsg);
    }
};
```

Run the container:

The service is implemented within an executable. At a minimum, we need to create within our main function an instance of the service container and run it.

Note that your service is started by parameters.

```
int main(int argc, char* argv[])
{
    // return value of our service program
    int retVal = 0;
    try
    {
        // Create the container and run it
        MyServiceContainer myContainer;
        myContainer.run();
    }
```

Catch exceptions:

Catch exceptions in case the container fails to start running.

```
catch(SoamException& exp)
    {
        // report exception to stdout
        cout << "exception caught ... " << exp.what() << endl;
retVal = -1;
    }
    // NOTE: Although our service program will return an overall
    // failure or success code it will always be ignored in the
    // current revision of the middleware.
    // The value being returned here is for consistency.
    return retVal;
}
```

CHAPTER

3

Developing Clients

Synchronous client structure

The following section provides the steps to create a synchronous client.

Summary of steps to develop a synchronous client

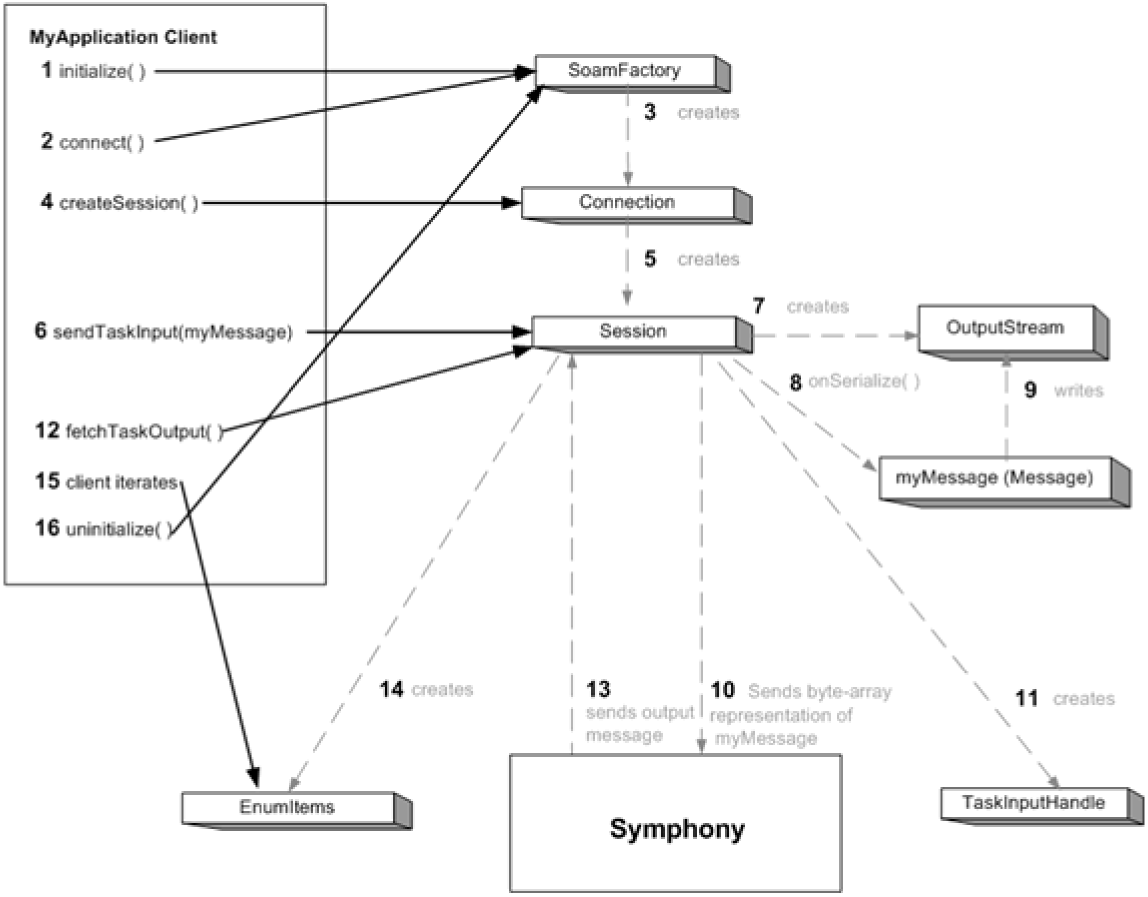
To create a synchronous client, you need the following API calls:

1. Initialize an application in Symphony with SoamFactory.initialize().
2. Get a connection to an application with `SoamFactory.connect()`.
3. Create a session with `Connection.createSession()`.
4. Send task inputs to the service in the application with `Session.sendTaskInput()` .
5. Retrieve task outputs with `Session.fetchTaskOutput()`.
6. Uninitialize the API with `SoamFactory.uninitialize()`.

Important:

Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

Synchronous client flow



1. The client calls the static `SoamFactory.initialize()` to initialize the API.

2. The client calls the static `SoamFactory.connect()` to establish a connection with Symphony.
3. The method creates and returns a `Connection` object that represents the physical connection to Symphony.
4. The client calls `Connection.createSession()`.
5. The method creates and returns a session object, which acts as a conduit through which the client can send input to its service.
6. The client creates an input message called myMessage and invokes `Session.sendTaskInput()` on the session object. This starts the chain of events that eventually sends the input to the service for processing.
7. The call to `sendTaskInput()` causes an `OutputStream` to be internally created.
8. The session calls `onSerialize()` on the input message and passes the `OutputStream` to the method.
9. In the `onSerialize()` method, the input message writes itself to the provided `OutputStream`.
10. The byte array representation of the input message is sent to Symphony.
11. As a result of the initial `sendTaskInput( )` call, Symphony returns a `TaskInputHandle` to the client, which contains an identifier that can help match the input to the output that will later return. At the same time, Symphony also sends the input message to the service for processing and obtains the output from the service invocation.
12. The client calls `fetchTaskOutput( )` method on the session object.
13. Symphony sends the output back to the client as a result of the `fetchTaskOutput( )` method call.
14. The output is put into an `EnumItems` object, which is basically a list of outputs, if multiple outputs are retrieved.
15. The client iterates over the `EnumItems` object to inspect each `TaskOutputHandle`. The `TaskOutputHandle` is a container for the output from the `Service`.
16. The client must call the static `SoamFactory.uninitialize( )` method to uninitialize the API.

Important:

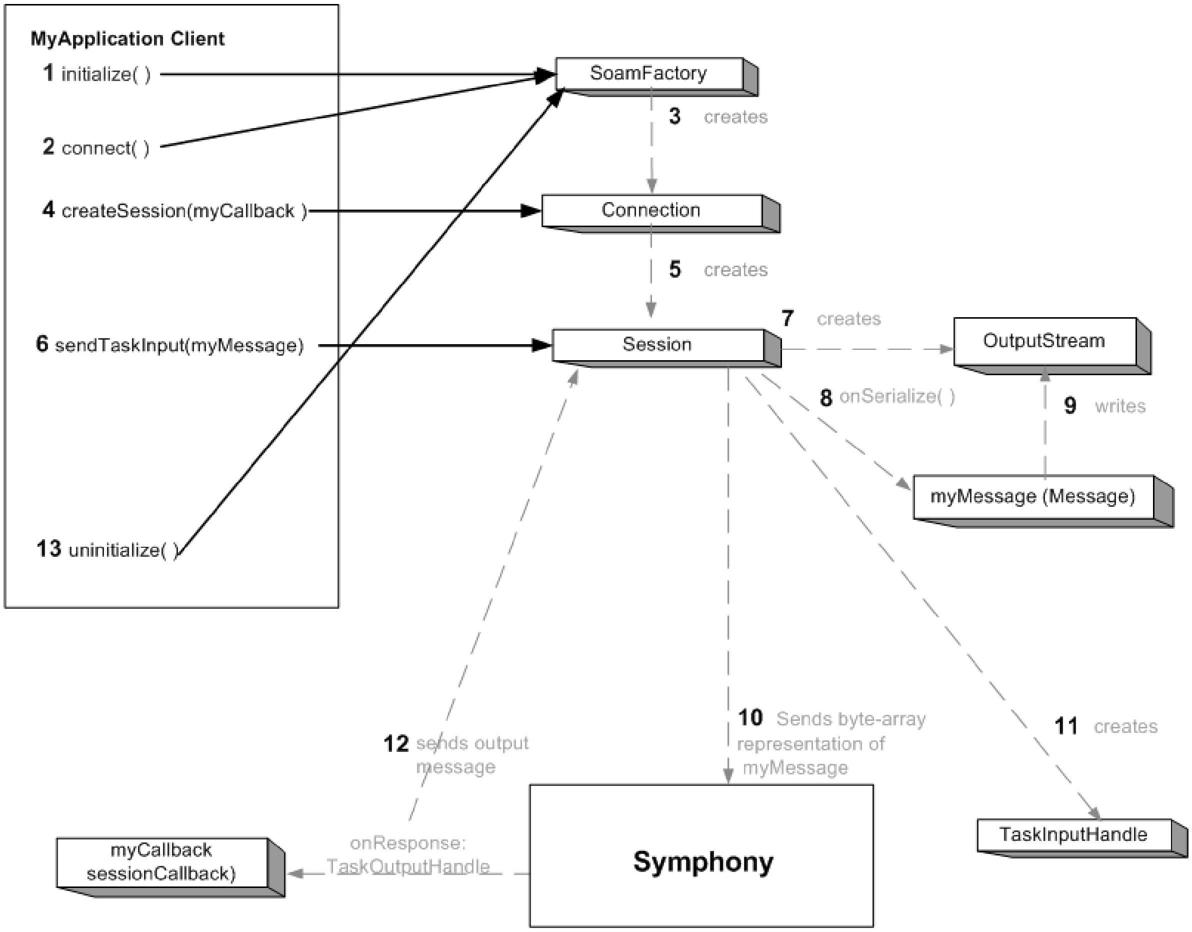
Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

Asynchronous client structure

The following section provides the steps to create an asynchronous client.

Asynchronous client flow

The flow for an asynchronous client is very similar to that of a synchronous client.



Differences in the flow for synchronous and asynchronous clients are highlighted below:

4. The client calls the `Connection.createSession( )` method on the `Connection` object and passes in a `SessionCallback` object to the method.

12. Symphony sends the output back to the client as soon as it is ready. A separate thread is used to invoke the `onResponse()` method on the callback that was provided at the time of session creation. The `onResponse()` receives a `TaskOutputHandle` argument that contains the output from the service.

SessionCallback

Used to receive events in an asynchronous manner. Events can take the form of results, exceptions, etc., that can occur within the scope of a session.

Note:

The SessionCallback must exist for the lifetime of the session. The callback cannot be destroyed until the session is closed.

Feature: On-demand results retrieval

The on-demand results retrieval feature enables a Symphony client to request results directly from the Symphony Session Manager in order to control the amount of memory consumed on the client host.

Scope

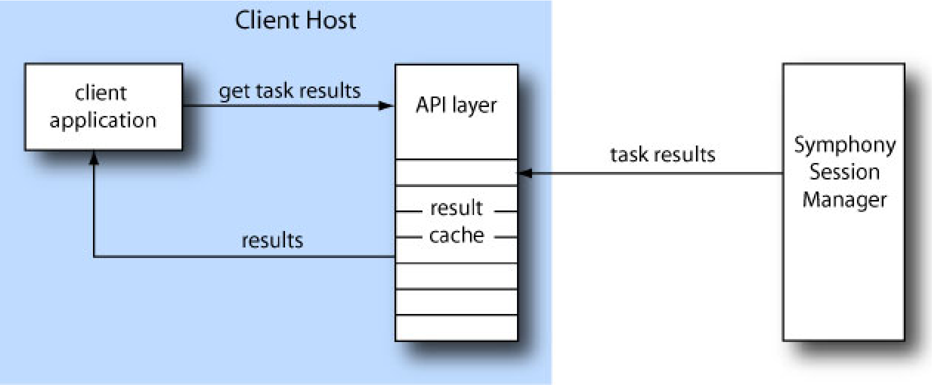
Operating system	• UNIX and Windows hosts
Limitations	• This feature only applies to clients that receive results synchronously. • This feature does not work consistently if the session type attribute discardResultsOnDelivery is set to false. This may cause every direct fetch of N items to return the same results.

About on-demand results retrieval

Before describing how on-demand results retrieval can impact memory management on the client host, it is helpful to know how results are processed in Symphony's default model.

Default behavior without on-demand results retrieval

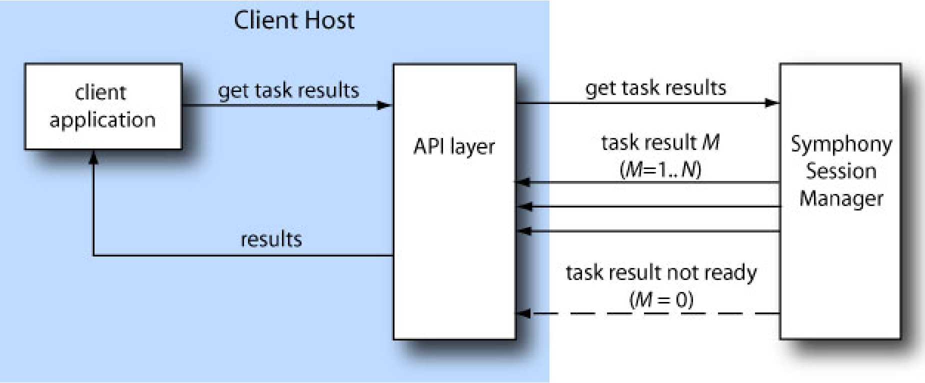
Since the underlying communication channel between the client and the Symphony Session Manager is asynchronous, results are sent to the client as soon as they are available. If the client makes a request for results before they arrive, the client code is blocked until the results arrive or the wait period expires. If the results arrive before the client makes a request, the results are queued (cached) in the local session's result set. Since the delivery of results from the Symphony Session Manager is not in sync with the retrieval of results, it is possible that too many results could be queued in the API layer; this could cause the client to run out of memory and eventually terminate abnormally.



Behavior with on-demand results retrieval

The purpose of the on-demand results retrieval feature is to control the retrieval of results so that the client is not overloaded with results and runs out of memory. This is achieved by having the client retrieve data directly from the Symphony Session Manager instead of from the local result cache. To better understand the interaction between the client and the Symphony Session Manager, let's look at the sequence of events.

1. Symphony Session Manager gets the result from the service and defers dispatching it to the client.
2. Client makes an API call to request the next result set from the Symphony Session Manager.
3. The Symphony Session Manager returns as much of the result set that it has available, i.e., 0-N, to the client.



Client API

The on-demand results retrieval feature can only be accessed through the client API. To use this feature, the client application must perform the following sequence:

1. Create a session using a flag to inform the API of the client's intent to fetch results directly from the Symphony Session Manager.
2. Send the task.
3. Fetch results in manageable amounts.

The session flag is a member of the SessionCreationAttributes class. The following list shows how the session flag is set for on-demand results retrieval in each supported language.

C++:	.setSessionFlags(Session::FetchResultsDirectly)
Java:	.setSessionFlags(Session.FETCH_RESULTS_DIRECTLY)
C#.NET:	.SessionFlags=SessionFlags.FetchResultsDirectly

Important:

Normally the Session.FetchTaskOutput(ulong countMax) method blocks indefinitely until countMax task responses are ready. When your session is created using the FetchResultsDirectly flag, the call does not block indefinitely but instead returns with 0 - N items. The number of items returned depends on the amount of results available in the Symphony Session Manager at the time of the call. Therefore, the client must poll until the desired number of results have returned.

If the Session.FetchTaskOutput(ulong countMax, long timeoutInSeconds) method is used while the on-demand results retrieval feature is enabled, the timeout parameter is ignored by the Symphony Session Manager.

Feature: Selectively Retrieving Task Results

Selective task output retrieval enables a Symphony client to request task results by specifying task IDs, in order to control the amount of memory consumed on the client host.

Scope

Operating system (client)	• All platforms supported by Symphony
Limitations	• This feature only applies to clients that receive results synchronously. • This feature is only available if the session is created with the `FetchResultsDirectly` session flag. • COM API is not supported. • Multi-threaded use of this feature is not recommended unless the client implements a synchronization scheme to prevent overlapping criteria from being specified in concurrently executing threads; otherwise the client may hang or falsely validate the task output retrieval.

About selective task result retrieval

Before learning how this feature can impact memory management on the client host, it might be helpful to know how results are processed in Symphony's default model.

Default task output retrieval

In the default model, the Symphony Session Manager sends results to the client as soon as they are available. If the client makes a request for results before they arrive, the client code is blocked until the results arrive or the wait period expires. If the results arrive before the client makes a request, the results are queued (cached) in the local session's result set. Since the delivery of results from the Session Manager is not in sync with the retrieval of results, it is possible that too many results could be queued in the API layer; this could cause the client to run out of memory and eventually terminate abnormally. One possible solution is to have the client request a specific number of results directly from the Session Manager; this feature is explained in *On-demand results retrieval* on page 44. This can help to manage memory usage on the client host but the requested results may not be the ones the client is really interested in.

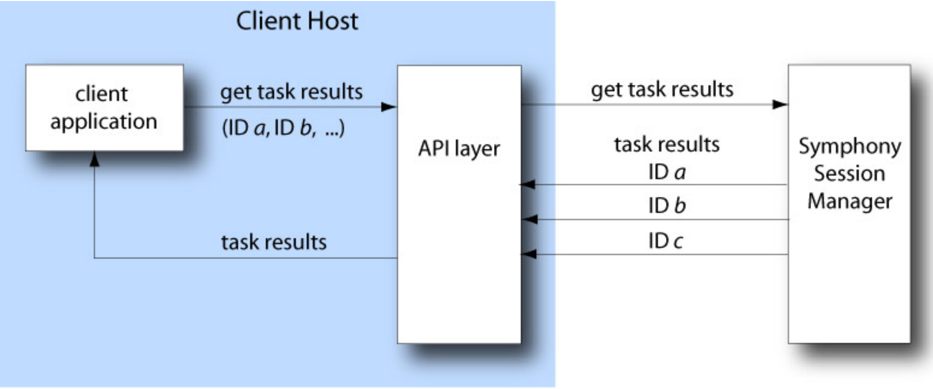
Symphony offers another option, which enables a client to request specific tasks results via selection criteria. Using this feature, the client gets only the results it wants.

Selective task output retrieval

The purpose of selective task output retrieval is twofold: it controls the retrieval of results so that the client is not overloaded with results and runs out of memory, and it allows the client to retrieve only the results it is interested in. This is achieved by having the client retrieve data directly from the Session Manager instead of from the local result cache. To better understand the interaction between the client and the Session Manager, let's look at the sequence of events.

1. Session Manager gets the result from the service and defers dispatching it to the client.
2. The client makes an API call to selectively retrieve task results from the Session Manager by supplying the task IDs for the results it wants to process.
3. In the non-blocking model (zero timeout), the Session Manager returns all the results (that match the task IDs) available at the time of the API call. If no matching task IDs are found, no results are returned.

In the blocking model (infinite timeout), the API call does not return until all the results with matching task IDs are available.



Note:

Task results returned to the client are not sorted. For example, if the client retrieves the output of tasks with IDs 1, 2, 3, the results may be returned in a different order.

Client API

Selective task results retrieval can only be achieved through the client API. To use this feature, the client application must perform the following sequence:

1. Create a session using the `FetchResultsDirectly` flag to inform the API of the client's intent to retrieve results directly from the Symphony Session Manager.
2. Send the tasks.
3. Retrieve results by specifying the associated task IDs in a selection filter.

Code samples for the blocking model

The following code samples demonstrate selective task result retrieval for the blocking model in each supported language. Refer to the API reference documentation in the Knowledge Center for more information.

```
//C++ sample
try
{
    ……
    // Set up session creation attributes and create the session
    attributes.setSessionFlags(Session::ReceiveSync|Session::FetchResultsDirectly);
    SessionPtr sesPtr = conPtr->createSession(attributes);
    //Create a task ID filter object
    TaskIdFilter filter;

    // Now we will send some messages to our service
    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
    {
        ……
        // send tasks
        TaskInputHandlePtr input = sesPtr->sendTaskInput(attrTask);
        // add specific task ID into task ID filter
        filter.addId(input->getId());
    }

    //Block until all filtered task outputs are ready.
    EnumItemsPtr enumOutput = sesPtr->fetchTaskOutput(filter);
    //handle the task outputs
    …….
}
```

```
//Java sample
try
{
    ……
    // Set up session creation attributes and create the session
    attributes.setSessionFlags(Session.RECEIVE_SYNC | Session.FETCH_RESULTS_DIRECTLY);
    session = connection.createSession(attributes);
    //Create a task ID filter object
    TaskIdFilter filter = new TaskIdFilter();

    // Now we will send some messages to our service
    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
    {
        ……
        // send tasks
        TaskInputHandle input = session.sendTaskInput(attrTask);
        // add specific task ID into task ID filter
        filter.addId(input.getId());
    }

    //Block until all filtered task outputs are ready.
    EnumItems enumOutput = session.fetchTaskOutput(filter);
    //handle the task outputs
    …….
}
```

```
//C# sample
try
{
    ……
    // Set up session creation attributes and create the session
    attributes.SessionFlags = SessionFlags.ReceiveSync |
    SessionFlags.FetchResultsDirectly;
    session = connection.CreateSession(attributes);
    //Create a task ID filter object
    TaskIdFilter filter = new TaskIdFilter();

    // Now we will send some messages to our service
    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
    {
        ……
        // send tasks
        TaskInputHandle input = session.SendTaskInput(attrTask);
        // add specific task ID into task ID filter
        filter.AddId(input.Id);
    }

    //Block until all filtered task outputs are ready.
    EnumItems enumOutput = session.FetchTaskOutput(filter);
    //handle the task outputs
    …….
}
```

Code samples for the non-blocking model

The following code samples demonstrate selective task result retrieval for the non-blocking model in each supported language. Refer to the API reference documentation in the Knowledge Center for more information.

```
//C++ sample
try
{
    ……
    // Set up session creation attributes and create the session
    attributes.setSessionFlags(Session::ReceiveSync|Session::FetchResultsDirectly);
    SessionPtr sesPtr = conPtr->createSession(attributes);
    //Create a task ID filter object
    TaskIdFilter filter;

    // Now we will send some messages to our service
    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
    {
        ……
        // send tasks
        TaskInputHandlePtr input = sesPtr->sendTaskInput(attrTask);
        // add specific task ID into task ID filter
        filter.addId(input->getId());
    }
    while (!filter.isSatisfied())
    {
        EnumItemsPtr enumOutput = sesPtr->fetchTaskOutput(filter,
        0 /* non-blocking */);
        //handle the task output
        …….
        // Since the current thread is not blocked waiting for all results, we can do
        // something else in between fetch attempts to make use of the current thread.
    }
}
```

```
//Java sample
try
{
    ……
    // Set up session creation attributes and create the session
    attributes.setSessionFlags(Session.RECEIVE_SYNC | Session.FETCH_RESULTS_DIRECTLY);
    session = connection.createSession(attributes);
    //Create a task ID filter object
    TaskIdFilter filter = new TaskIdFilter();

    // Now we will send some messages to our service
    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
    {
        ……
        // send tasks
        TaskInputHandle input = session.sendTaskInput(attrTask);
        // add specific task ID into task ID filter
        filter.addId(input.getId());
    }
    while (!filter.isSatisfied())
    {
        EnumItems enumOutput = session.fetchTaskOutput(filter, 0 /* non-blocking */);
        //handle the task output
        …….
        // Since the current thread is not blocked waiting for all results, we can do
        // something else in between fetch attempts to make use of the current thread.
    }
}
```

```
//C# sample
try
{
    ……
    // Set up session creation attributes and create the session
    attributes.SessionFlags = SessionFlags.ReceiveSync |
    SessionFlags.FetchResultsDirectly;
    session = connection.CreateSession(attributes);
    //Create a task ID filter object
    TaskIdFilter filter = new TaskIdFilter();

    // Now we will send some messages to our service
    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
    {
        ……
        // send tasks
        TaskInputHandle input = session.SendTaskInput(attrTask);
        // add specific task ID into task ID filter
        filter.AddId(input.Id);
    }

    while (!filter.IsSatisfied)
    {
        EnumItems enumOutput = session.FetchTaskOutput(filter, 0 /* non-blocking */);
        //handle the task output
        …….
        // Since the current thread is not blocked waiting for all results, we can do
        // something else in between fetch attempts to make use of the current thread.
    }
}
```

Security

Middleware security

The middleware relies on an external and extensible security infrastructure based on plugins.

Default security implementation

There is a default security plugin that works a user name-password pair for authentication against the Symphony user database.

In Symphony, you define users in the Platform Management Console. When a client application connects to Symphony, it provides a user account and password. The system checks the provided user account name and password against the Symphony user database.

The callback object DefaultSecurityCallback works together with the default plugin to meet very basic security requirements.

Custom security plugins

The use of the security framework is optional. Your security specialist can write a custom plugin to handle token generation and server component authentication. This must be coupled with an implementation of a callback object, which provides the required information on demand, such as for example, user name, password, eye scan, etc.

Security in Symphony DE

There is no security in Symphony DE.

SecurityCallback

Used in relation to events occurring within the scope of a connection. The `SecurityCallBack` is used when information is required for authentication.

If this callback has a lifetime that is less than that of the object it is associated with, the client API must be informed of its pending destruction to prevent it from attempting to invoke methods on the callback after its destruction. Inform the client API by calling `invalidateCallback()` on the respective associated objects. If you are not sure of the scope, make these calls explicitly to prevent unexpected errors within the client.

Ensure you keep the DefaultSecurityCallback as a global variable. The callback used for security has to exist for the lifetime of the connection.

You need to create the callback object on the heap and treat it the same way the SessionCallback is treated in respect to the session (but for the connection instead).

Connections

About connections

For a client to submit workload, it connects to an application and interacts with a session created on this connection.

The API binds this logical connection to an actual physical connection, which uses a socket between the client and Symphony. Multiple concurrent logical connections within the same client are multiplexed on a single physical connection.

Number of file descriptors opened per connection

One socket per client connection to a session manager.

All communications between the client and session manager are multiplexed on a single connection.

Length of time the connection is maintained

The client maintains a persistent connection to the session manager. This connection exists until the client explicitly closes the connection or terminates.

Setting client reconnection timeout

By default, the API attempts to refresh the connection between the client and the system if a client abruptly disconnects. If the attempt fails, the API throws an exception.

To control the way a client reconnects, set the following environment variables on the client machine:

- SOAM_RECONNECTION_RETRY_INTERVAL
- SOAM_RECONNECTION_RETRY_LIMIT
- SOAM_RELOCATED_RECONNECTION_RETRY_INTERVAL
- SOAM_RELOCATED_RECONNECTION_RETRY_LIMIT

For more details on these environment variables, see the *Symphony Reference*.

Sessions

About sessions

You create a local session object that refers to a session in Symphony.

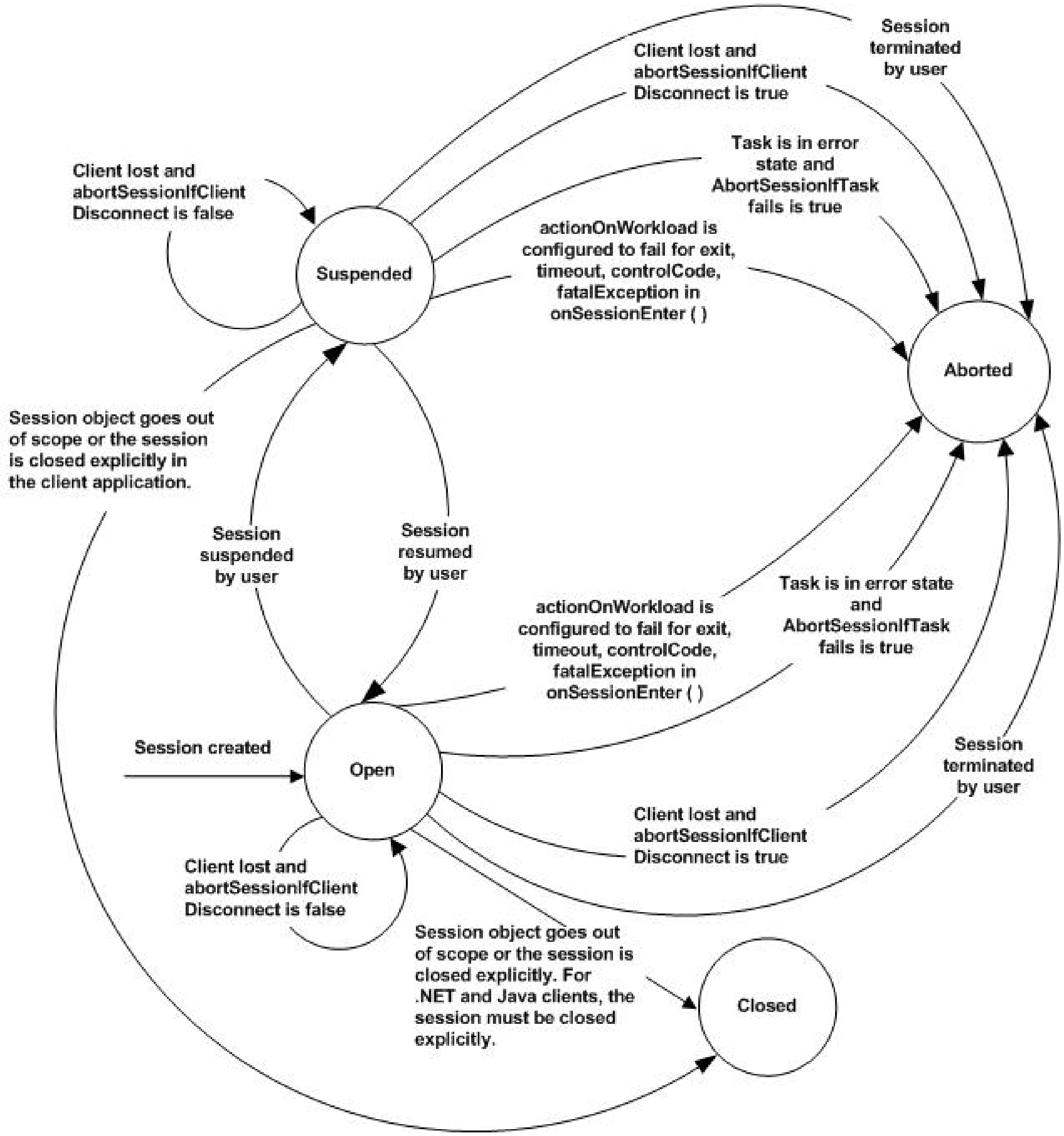
You can interact with a Symphony session by invoking methods on your local `Session` object.

Once the local session is created by the client, its corresponding Symphony session is valid until one of the following happens:

- The session is killed by an administrator
- The session aborts because of a fatal exception
- The client goes away without closing the session
- The client closes the session

Session lifecycle

In your client application, you can create a local `Session` object that refers to a session in Symphony. You can interact with Symphony session by invoking methods on your local `Session` object.


Session terminated by user
Client lost and abortSessionIfClient Disconnect is true
Task is in error state and AbortSessionIfTask fails is true
Client lost and abortSessionIfClient Disconnect is false
Suspended
actionOnWorkload is configured to fail for exit, timeout, controlCode, fatalException in onSessionEnter ( )
Aborted
Session object goes out of scope or the session is closed explicitly in the client application.
Session suspended by user
Session resumed by user
actionOnWorkload is configured to fail for exit, timeout, controlCode, fatalException in onSessionEnter ( )
Task is in error state and AbortSessionIfTask fails is true
Session created
Open
Session terminated by user
Client lost and abortSessionIfClient Disconnect is true
Client lost and abortSessionIfClient Disconnect is false
Session object goes out of scope or the session is closed explicitly. For .NET and Java clients, the session must be closed explicitly.
Closed

Using tags for related sessions

Overview

A tag is simply a string that is attached to the session when it is created. Since the running of a functional job can involve multiple sessions, a session tag that is shared among sessions provides the ability to query or control these related sessions with a single action.

This functionality not only allows you to change the priority of current sessions that share the same session tag but can also extend the priority change to future sessions. If required, this priority change for future sessions can be reset so that the priority is derived from the application profile.

Session tag APIs

The client application is responsible for generating the session tag. The tag, which can be up to 128 characters in length, is limited to alphanumeric, hyphen, and underscore characters. If it is set as an empty string, the session will not have a session tag.

The session tag is set in the `SessionCreationAttributes` object, which is passed as an input parameter to the `createSession()` method. The following code samples demonstrate the use of `SessionCreationAttributes` for each of the supported programming languages. A VBA sample is also included for the COM API. For more details, refer to the API reference documentation.

C++

```
SesssionCreationAttributes attributes;
attributes.setSessionName("mySession");
attributes.setSessionType("mySessionType");
attributes.setSessionFlags(SF_SYNC);
attributes.setSessionTag("tag");
SessionPtr sesPtr = conPtr->createSession(attributes);
```

Java

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.setSessionName("mySession");
attributes.setSessionType("mySessionType");
attributes.setSessionFlags(Session.SYNC);
attributes.setSessionTag("tag");
session = connection.createSession(attributes);
```

C# (.NET)

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.SessionName="mySession";
attributes.SessionType="mySessionType";
attributes.SessionFlags = SessionFlags.AliasSync;
attributes.SessionTag="tag";
session = connection.CreateSession(attributes);
```

VBA (COM API)

```
Set attributes = New CSoamSessionCreationAttributes
attributes.SessionName="mySession"
attributes.SessionType="mySessionType"
attributes.SessionFlags = SessionFlags.ReceiveSync
attributes.SessionTag="tag"
Set session = connection.CreateSession(attributes)
```

Querying and controlling related sessions

Actions that query and control related sessions can be performed by supplying the session tag via the command line interface. Refer to the *Symphony Reference* for more information on the commands that accept the session tag as a filter.

The session tag can also be supplied through the Platform management console as a filter. Refer to the online documentation provided with the management console for further information.

Feature: Session-creation attributes

Client-side features of the `SessionCreationAttributes` API enable developers to dynamically specify the priority and the associated service while creating a session.

Scope

Applicability	Details
Operating system	• Windows • Linux • Solaris
Limitations	None

About client-side session creation workload attributes

The following attributes can be specified at session creation. You can apply SessionPriority and ServiceName separately, together, or not at all.

ServiceName

Any service name that exists in the application profile. Workload submitted to the session is sent to the service specified in the `SessionCreationAttributes`.

If session type is specified, the configured value for ServiceName in the session type will be used as default.

If session type is not specified when creating the session, the default ServiceName is the service configured as default="true" in the application profile.

SessionFlags

Flags that control how the API will interact with the session.

SessionName

Text to be associated with the session. This name does not have to be unique and exists to give informational details as desired by the creator of the session. SessionName is limited to 256 characters.

SessionPriority

A priority between 1 and 10,000.

If session type is specified, the configured value for SessionPriority in the session type will be used as default and ServiceName.

If no value is configured for SessionPriority, the default SessionPriority of 1 is used.

If session type is not specified when creating the session, the default SessionPriority is 1.

SessionTag

A text to be associated with a session that can be used in administrative operations. SessionTag is limited to 128 characters.

SessionType

A name associated with a collection of other attributes that can be assigned to a session on its creation. The session type specified with the API must match the session type that is defined in the application profile.

The session type is optional. If you leave this parameter blank " " or do not set a session type, system default values are used for session attributes. If you specify a session type in the client application, you must also configure the session type in the application profile—the session type name in your application profile and session type you specify in the client must match. If you use an incorrect session type in the client and the specified session type cannot be found in the applicatin profile, an exception is thrown to the client.

Example: Set workload session attributes with SessionCreationAttributes API

To set a priority of 100 and use the service named calc service associated with the session named `My Session`, your session creation code would look like this:

C++

```
SessionCreationAttributes attrs;
attrs.setSessionPriority(100);
attrs.setServiceName("calc");
attrs.setSessionName("My Session");
attrs.setSessionType("LongRunningSession");
attrs.setSessionFlags(ReceiveSync);
attrs.setSessionTag("S9v234");
...
```

C# (.NET)

```
SessionCreationAttributes attrs = new SessionCreationAttributes();
attrs.SessionPriority = 100;
attrs.ServiceName="calc";
attrs.SessionName="My Session";
attrs.SessionType="LongRunningSession";
attrs.SessionFlags = ReceiveSync;
attrs.SessionTag="S9v234";
...
```

Java

```
SessionCreationAttributes attrs = new SessionCreationAttributes();
attrs.setSessionPriority(100);
attrs.setServiceName("calc");
attrs.setSessionName("My Session")
attrs.setSessionType("LongRunningSession");
attrs.setSessionFlags(ReceiveSync);
attrs.setSessionTag("S9v234");
...
```

Tasks and messages

Tasks

A task is simply a container to hold matching input and output. You cannot directly create a task. However, you can submit work to a session by invoking `sendTaskInput( )`, which creates a task in Symphony. No output object exists until the service instance returns the result to Symphony after processing the input.

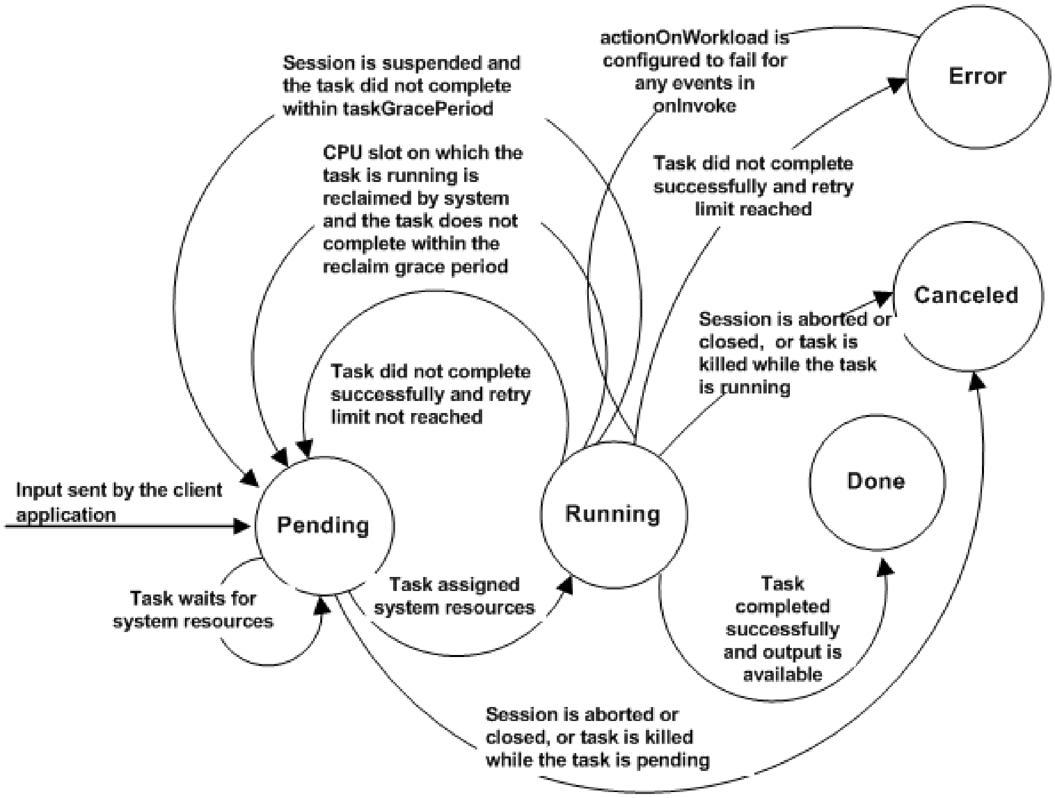
Task lifecycle

Once a task is created, it is valid in Symphony until one of the following conditions exist:

- The client collects the output and Symphony receives confirmation that the output was successfully collected
- The session is terminated, which in turn terminates all tasks
- The task is killed

You can specify that if a task fails it should be rerun by specifying the taskRetryLimit attribute in the application profile. This informs Symphony how many times it should rerun the task before giving up.

If a task fails, Symphony attempts to rerun the task up to the number of times specified by the taskRetryLimit attribute in the application profile. If the task has not successfully run after *n* retries, and the abortSessionIfTaskFail attribute is set true in the application profile then the session is aborted. Otherwise, a single task failure does not affect the session. By default, abortSessionIfTaskFail = false.


actionOnWorkload is configured to fail for any events in onInvoke
Error
Session is suspended and the task did not complete within taskGracePeriod
CPU slot on which the task is running is reclaimed by system and the task does not complete within the reclaim grace period
Task did not complete successfully and retry limit reached
Canceled
Session is aborted or closed, or task is killed while the task is running
Task did not complete successfully and retry limit not reached
Done
Input sent by the client application
Pending
Running
Task waits for system resources
Task assigned system resources
Task completed successfully and output is available
Session is aborted or closed, or task is killed while the task is pending

Task tags: Using tags for related tasks

Overview

A tag is simply a string that is attached to the task when it is created. Since the tasks for each session can come from different user requests, the tasks that belong to a particular request can be identified by their tags. The task tag can later be used to filter only those tasks that share the given tag.

Task tag APIs

The client application is responsible for generating the task tag. The tag, which can be up to 128 characters in length, is limited to alphanumeric, hyphen, and underscore characters. Note that the task tag cannot begin with a hyphen.

The task tag is set in the `TaskSubmissionAttributes` object, which is passed as an input parameter to the `sendTaskInput()` method. The following code samples demonstrate the use of the task tag for each of the supported programming languages. For more details, refer to the API reference documentation.

C++

```
// Create a message
char hello[] = "Hello Grid !!";
MyMessage inMsg(taskCount, true, hello);
// Create task attributes
TaskSubmissionAttributes attrTask;
attrTask.setTaskInput(&inMsg);
attrTask.setTaskTag("TaskTagHere");
// send it
TaskInputHandlePtr input = sesPtr->sendTaskInput(attrTask);
```

Java

```
// Create a message
MyInput myInput = new MyInput(taskCount, "Hello Grid !!");
// Set task submission attributes
TaskSubmissionAttributes taskAttr = new TaskSubmissionAttributes();
taskAttr.setTaskInput(myInput);
taskAttr.setTaskTag("TaskTagHere");
// Send it
TaskInputHandle input = session.sendTaskInput(taskAttr);
```

C# (.NET)

```
// Create a message
MyMessage inputMessage = new MyMessage(taskCount, true, "Hello Grid !!");
// Set task submission attributes
TaskSubmissionAttributes taskAttr = new TaskSubmissionAttributes();
taskAttr.SetTaskInput(inputMessage);
taskAttr.TaskTag = "TaskTagHere";
// Send it
TaskInputHandle input = session.SendTaskInput(taskAttr);
```

Querying and displaying related tasks

Actions that query and display related tasks can be performed by supplying the task tag or wildcard characters via the command line interface. Refer to Symphony Reference documentation for more information on the commands that accept the task tag as a filter.

The task tag can also be supplied through the Platform management console as a filter. Refer to the online documentation provided with the management console for further information.

Priority tasks

When you send input messages, they are used to create tasks.

- If there are enough CPU slots to compute the tasks, tasks are immediately sent to compute hosts.
- If there are not enough CPU slots to compute the tasks, tasks remain pending in a dispatch queue according to submission time. Tasks are sent to compute hosts in the order in which they were submitted.

In some cases, when tasks are pending, you may want to have a task jump to the start of the queue and be sent to compute hosts before other tasks. You can do this by specifying that a task has priority.

Running tasks are not preempted for priority tasks.

When one task has priority

You specify that a task has priority when you send the message.

If more than one task in a session is a priority task, subsequently submitted priority tasks jump ahead of pending priority tasks that are still waiting to be sent to compute hosts.

If there are pending tasks, the priority task jumps the queue so that it is sent to a compute host for processing ahead of other tasks in the queue.

Messages

Input and output messages

The `Message` class is an object-oriented construct that enables you to send your data through Symphony to your client or service.

To send custom input and output, the `Message` object must be extended and the `onSerialize( )` and `onDeserialize( )` methods implemented.

The Symphony API follows object-oriented principles to allow your data to be logically encapsulated.

Input messages

Any input to the service that does not fall under the category of common data is referred to as an *input message*.

An input message must be sent to invoke your service. Each input message corresponds to one service invocation.

You can determine the scope of an input message retrieved by the `ServiceContainer` by accessing the input message during the execution of `onInvoke( )` on the `ServiceContainer`

Lifetime of an input message

The input message sent from the client cannot be deleted until `sendTaskInput( )` on the session returns successfully.

If you have a recoverable session: once `sendTaskInput( )` returns successfully, the input message is kept in the system until output is successfully retrieved for the task. If you do not retrieve output successfully, the message is kept in the system until the session is closed or aborted.

If you have a non-recoverable session: if the session manager fails or the system goes down, you need to keep your input message and resend it, or create a new session and resend. You will not be able to recover input or output.

Output messages

The result of a service computation to be sent back to the client is referred to as an *output message*.

Lifetime of an output message

The output message sent from the service cannot be deleted until `setOutputMessage( )` on the `TaskContext` returns successfully.

Data passing: serialization and deserialization

In the Symphony API, you need to implement methods to read messages from I/O streams. Symphony handles the serialization and deserialization of messages between the client application and the service.

Serialization

Input to the `onSerialize( )` method is an empty output stream. You must write the code that puts your data into that stream.

Deserialization

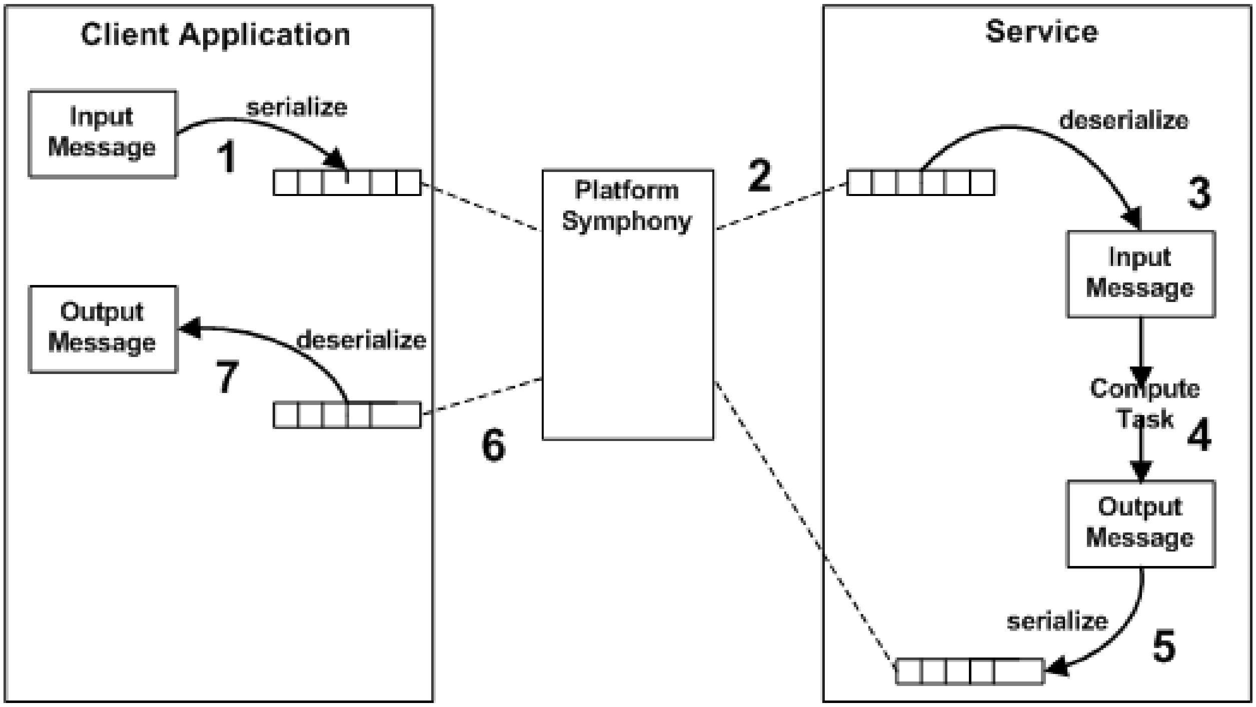
Input to the `onDeserialize( )` method is an output stream that contains your serialized data. You must write the code to populate your message's member variables with the data from the stream.

Tip:

Remember to double-check that your `Message` object serialization and deserialization order are the same.

How messages are serialized and deserialized in the system: a high-level overview

1. When the client calls the appropriate API method to send a task input to the service, the input message is serialized to a binary form.
2. Symphony transports this binary data to the service with no knowledge of its content.
3. In order to do its work, the service needs to gain access to the input message. The service calls the appropriate API method to access the input message, which causes the binary data to be deserialized to an input message.
4. The service processes the input message and generates a processing result (an output message).
5. The service sets this output message to indicate its intent to return the result to the client. When the service calls the appropriate API method to set the output, the output message is serialized to a binary form.
6. When the service processing method returns, Symphony transports this binary data to the client with no knowledge of the data's content.
7. The client calls the appropriate API method to access the output message from the service, which causes the binary data to be deserialized to an output message. The client then proceeds to process the output.



Feature: Default message API

The default message API gives developers quicker integration points to help shorten the integration effort with Symphony for those applications which might have heavy dependence on strings or raw binary data being transferred between the client and service. Developers now do not need to implement `onSerialize()` and `onDeserialize()` when sending messages composed of a single text string or a stream of binary data.

Scope

Applicability	Details
Operating system	• Windows • Linux • Solaris
Limitations	None

About default message classes

The following message objects give developers instant access to message implementations that are available out of the box. Both of these objects can be used within cross-language applications to implement clients and services in different Symphony API targets.

DefaultTextMessage	Contains basic implementation to encapsulate a string within a message object. This means the developer is able to easily transfer a single text string between client and service by having the application call the setText and getText methods.

DefaultByteArrayMessage	Contains basic implementation to encapsulate a byte array within a message object. This means the developer is able to easily transfer binary data between client and service by having the application call the setByteArray and getByteArray methods.

Modifying your client for performance

Feature: Improving throughput in high-latency networks

Aggregating messages prior to putting them on the wire can improve throughput in high-latency networks such as WANs. This technique enables a Symphony client to maximize utilization of the network connection between itself and the session manager.

Scope

Applicability	**Details**
Operating system	• Windows • Linux • Solaris
Best practice	This feature is not recommended for recoverable clients since message aggregation would require complex application coding to handle a client that terminates abnormally. The reason that it is not recommended to use message aggregation with recoverable clients is that due to message aggregation, the number of outstanding sends to the session manager is greater than 1. As a result, there is no way to know how many outstanding sends were successfully received or are in the process of being handled by the session manager. A recoverable client is a client that can tolerate an abnormal termination of its execution and is able to recover and continue to process workload. Recovery of such a client usually involves it being restarted and given enough context to allow it to connect and open an existing session that previously contained its workload. For this type of client, it is usually recommended to set the discardResultsOnDelivery attribute to “false” in the applicaton profile to allow for a simplified recovery procedure.

About message aggregation

Before describing how message aggregation can improve throughput for a WAN connection, it is helpful to know how task data is submitted in Symphony’s default model.

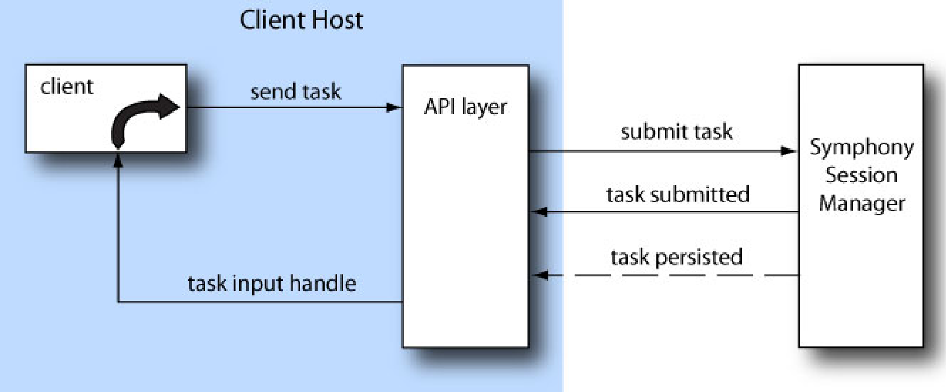
Default behavior without message aggregation

Although the underlying communication channel between the client and the session manager is asynchronous, the data submission protocol is synchronous. Here is the sequence of events when a client wants to send task data to the session manager:

1. Client sends task data to the API layer.
2. API layer serializes the data and submits it to the underlying communication layer.
3. API layer blocks the client’s submission thread.
4. Data is transferred by the communication layer to the session manager. The session manager replies with an acknowledgement upon successful receipt of the data.
5. API layer returns a Task Input Handle to the client and unblocks the client’s thread. The client’s thread is then free to submit more input.

Note:

If the session’s recoverable attribute is set to true, the API waits for confirmation of successful data persistence from the session manager before returning a Task Input Handle to the client.



Behavior with message aggregation

The message aggregation feature enables the submission of work in a non-blocking mode. To better understand the interaction between the client and the session manager when using message aggregation, let’s look at the sequence of events.

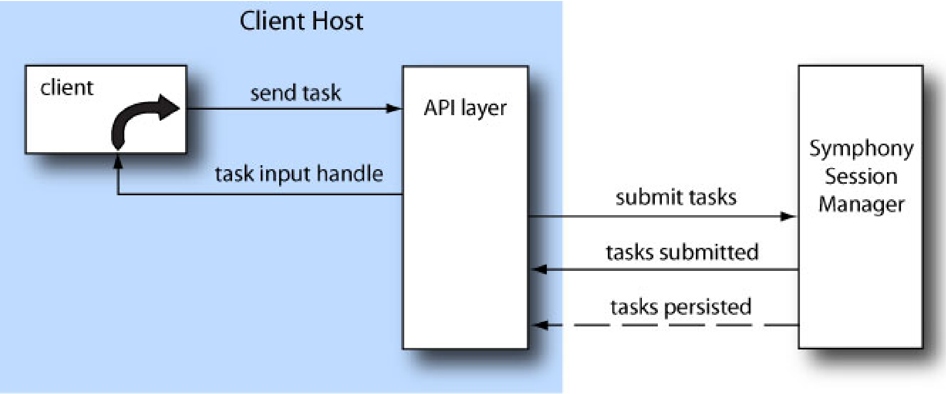
1. Client sends task data to the API layer.
2. API layer serializes the data and submits it to the underlying communication layer.
3. API layer returns a Task Input Handle to the client and program execution returns to the client’s submission thread. The client repeats the data submission process until all the data is submitted.
4. Client waits with a collection of Task Input Handles until the session manager confirms that all the data has been received.

Note:

The task ID is only set in the Task Input Handle after the message has been successfully delivered to the session manager, i.e., the client has received confirmation from the session manager.

Note:

If the session’s recoverable attribute is set to true, the API waits for confirmation of successful data persistence from the session manager.

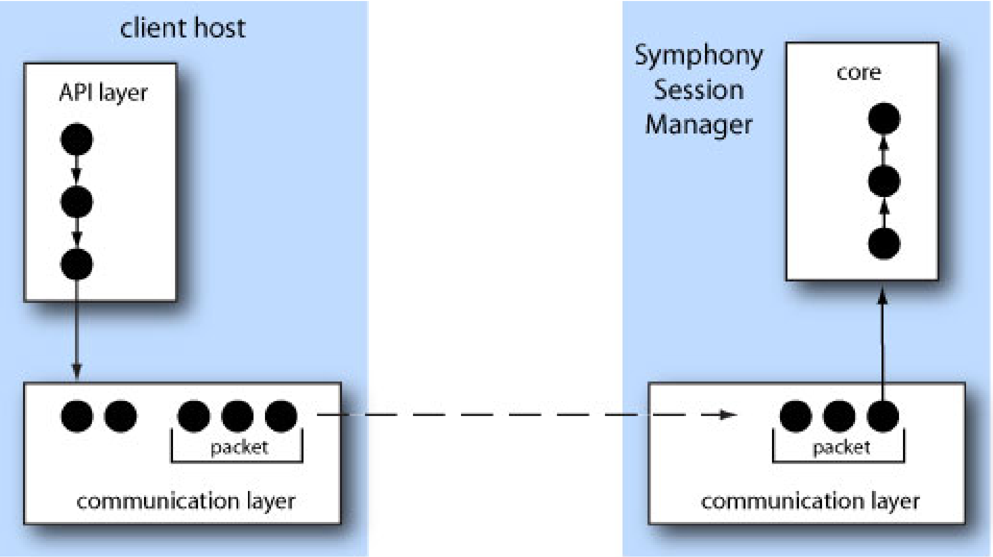


When deploying message aggregation, the client’s sendTaskInput() call becomes non-blocking and returns with a valid Task Input Handle object as soon as the data to be sent is queued for dispatch in the underlying communication layer. Once the call returns, the client can use the returned Task Input Handle to synchronize and/or query the state of the submission to the session manager. Since the communication layer dispatches as many queued items as it can fit into a dispatch unit, it means that data is aggregated transparently to the client code.

Although operations between the client and the session manager are asynchronous, they are still serialized and have no priority associated with them. This means that the order in which the operations are dispatched is always preserved. As a consequence, if a "direct fetch" or "close" operation is issued immediately after the client sends many tasks in rapid succession (as is the case with message aggregation), it is unlikely that the task queue will be empty. The "direct fetch" or "close" operation will be queued accordingly and dispatched in due time. As a result, the client may experience a delay for the operation in the queue to complete if there are many other pending operations at the time of the call.

Dispatch unit aggregation

In WAN networks, since the packet-level turnaround time is longer than for local network connections, the connection can be better utilized by packing as much data as possible into a packet before sending it. This is achieved by setting the TCP_NODELAY attribute in the `SD.xml` configuration file; refer to Configurable TCP connection attributes.



Client API

The message aggregation feature can only be accessed through the client API.

Client requirements

To enable message aggregation, the client application must do the following:

1. Create a session using the appropriate session flag to inform the API of the client’s intention to send task data in an overlapped manner.

   The session flag is a member of the SessionCreationAttributes class. The following list shows how the session flag is set for overlapped sending in each supported language.

C++:	.setSessionFlags(Session::SendOverlapped)
Java:	.setSessionFlags(Session.SEND_OVERLAPPED)
C#.NET:	.SessionFlags=SessionFlags.SendOverlapped

2. Perform the send operation using `sendTaskInput()`.
3. Since the send operation may be pending, the client must keep track of the returned Task Input Handle for verification and/or synchronization at a later time.
4. Client must query the Task Input Handle to evaluate whether the data has been successfully submitted to the session manager. The query may take the form of issuing a wait() call with no timeout or the client may poll for an update in the status.

   Wait calls have the following possible outcomes:

   - Data was successfully submitted so the wait call returns with a success code.
   - Data was unsuccessfully submitted so the wait call returns with a failure code, at which point the client must acquire the exception from the Task Input Handle to find out the reason for failure. If the optional parameter of throwOnSubmissionFailure is set to true, the method throws the exception directly. The default behavior is not to throw an exception.
   - The wait timed out and the call returns to the client.

- The wait call may also throw an exception if there is an internal error while performing the wait operation.

For more information about the wait methods of the `TaskInputHandle` class, refer to the API reference documentation.

Configurable TCP connection attributes

When configuring a remote client using large latency WAN connections, performance may be improved by setting TCP_NODELAY=0 and applying other appropriate TCP settings (based on user-specific network parameters to optimize TCP package throughput over the connection). Refer to Configuring TCP Connections.

Feature: Data compression

Data compression enables a client to improve data throughput in the network by reducing the size of data it sends to Symphony. The data size threshold can be specified by the application at runtime.

Scope

Applicability	**Details**
Operating system	Windows Linux Solaris
Limitations	N/A

About data compression

By default, data compression is not enabled. Once a session is created with data compression enabled, it is preserved for the lifetime of the session and cannot be changed by opening or updating the session.

Data compression can be used in conjunction with the direct data transfer feature. In this case, compressed data will be sent to and from the service.

During the compression/decompression step, the client and service will consume additional memory, as required, to complete the compression/decompression operation. Once compression/decompression has completed, the additional memory will be released.

When to use data compression

If your data is highly compressible and over 1Kb in size, you can experience an improvement in data throughput by using data compression. Highly compressible data is characterized as mostly text or a mixture of text and some binary. Tests have shown that a compression ratio between 85% and 95% can be achieved for highly compressible data. Since these ratios depend heavily on the makeup of the data, data analysis and trial runs of different messages within the application can be performed to get a better idea of the compression ratio for your specific application’s data.

The compression ratio of application data is determined by the following formula:

compression ratio = (1- compressed data size/uncompressed data size) x 100

Enabling data compression for sessions

When data compression is enabled, all task inputs and outputs, as well as all common data and common data updates can be compressed. Whether the data is compressed depends on the threshold setting.

Here is the sequence for compressing data at the session level and submitting it to Symphony.

1. The client creates a connection to Symphony.
2. The client creates a session and sets the session attributes so that data compression is enabled.
3. The client submits tasks using the session it has created. Data is sent to Symphony in compressed format if the data size exceeds the threshold setting.

Client API

The data compression feature can only be enabled through the client API.

Enabling data compression for sessions

You can enable data compression for all input/output tasks associated with a session, or for sending common data or common data updates. To enable data compression at the session level, the client application must do the following:

1. Create a session using the appropriate session attribute to inform the API of the client's intention to send data to Symphony in a compressed format. The session attribute is a member of the SessionCreationAttributes class.
2. Send the task input messages to Symphony.

The following sample code shows how a SessionCreationAttributes object is set for data compression in each supported language. For more information about the SessionCreationAttributes and SessionOpenAttributes classes, refer to the API reference documentation.

C++

```
SesssionCreationAttributes attributes;
attributes.enableDataCompression(true);
```

Java

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.enableDataCompression(true);
```

C# (.NET)

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.EnableDataCompression = true;
```

Setting the data compression threshold

When data compression is enabled, the threshold specifies the message size in kilobytes that triggers compression. As long as the byte size of the message is below the threshold, the data will be sent according to the default Symphony model. If the byte size is greater than the threshold, the data will be sent in a compressed format. The default threshold is 1 kilobyte.

Note:

In general, empirical data has shown that a threshold of less than 1K does not yield reasonable compression ratios. In fact, if your threshold is too low, you can experience negative compression ratios, depending on the type of data.

To set the data compression threshold, the client application must do the following:

1. Create a session using the appropriate session attribute to set the data compression threshold. The session attribute is a member of the SessionCreationAttributes class.
2. Send the task input message with the task attributes.

The following sample code shows how to set the data compression threshold with a SessionCreationAttributes object in each supported language. For more information about the SessionCreationAttributes class, refer to the API reference documentation.

C++

```
SessionCreationAttributes attributes;
attributes.setDataCompressionThreshold(5);
```

Java

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.setDataCompressionThreshold(5);
```

C# (.NET)

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.DataCompressionThreshold = 5;
```

Setting the data compression flag

The following flags offer the ability to further modify data compression behavior when compression is enabled. The flags are mutually exclusive.

- BetterSize: will give good compression in a reasonable time.
- BestSpeed: performs compression but considering time as a factor. This will yield a smaller compression ratio but will give the best compression time.

The default flag setting is BestSpeed.

Here is a summary of compression characteristics based on the analysis of empirical data:

- The time to decompress data will always be a fraction of the time it takes to compress the data. Depending on the compression flag selected, the time to decompress can be between 5% and 50% of the time to compress.
- For data containing mostly text (such as XML), once the data size is greater than 20KB, a compression ratio above 85% is expected. As the data size increases, the compression ratio approaches 95%.
- For data containing a mixture of text and binary formats, the average compression ratio, once the data size is greater than 1KB, is between 55% and 60%.
- The option to choose BestSpeed compression (default) or not can make a difference for highly compressible data, i.e., mostly text. In general, BestSpeed provides ratios that are about 4% lower than BetterSize in cases where data is highly compressible, but consumes noticeably less time (between 0.5 and 0.75 of the time it takes to compress with BetterSize).

The following sample code shows how to set the compression flags with a SessionCreationAttributes object in each supported language. For more information about the SessionCreationAttributes, refer to the API reference documentation.

C++

```
SesssionCreationAttributes attributes;
attributes.setDataCompressionFlags(Session::BestSpeed);
```

Java

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.setDataCompressionFlags(DataCompressionFlags.
BEST_SPEED);
```

C# (.NET)

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.DataCompressionFlags = DataCompressionFlags.BestSpeed;
```

Feature: Direct Data Transfer

Direct data transfer enables a Symphony client to maximize utilization of the network bandwidth between itself and the service. This feature essentially eliminates the session manager from the data flow allowing applications to optimize the use of different network topologies.

Scope

Applicability	Details
Operating system	• Windows • Linux • Solaris • AIX (client)
Limitations	• This feature cannot be used by offline clients since clients using the direct data transfer feature must always be available to provide data to the service instance while there is outstanding workload. • This feature cannot be used by recoverable clients. When the direct data transfer feature is enabled, data is actually cached in the running client instance instead of being sent to the session manager. There is no client-side recovery capability for this cached data. • This feature cannot be used with the Service Replay Debugger feature. • This feature cannot be used when the following session type attributes are set to the specified values, otherwise an exception is thrown. ◦ abortSessionIfClientDisconnects=”false” ◦ discardResultsOnDelivery=”false”

About direct data transfer

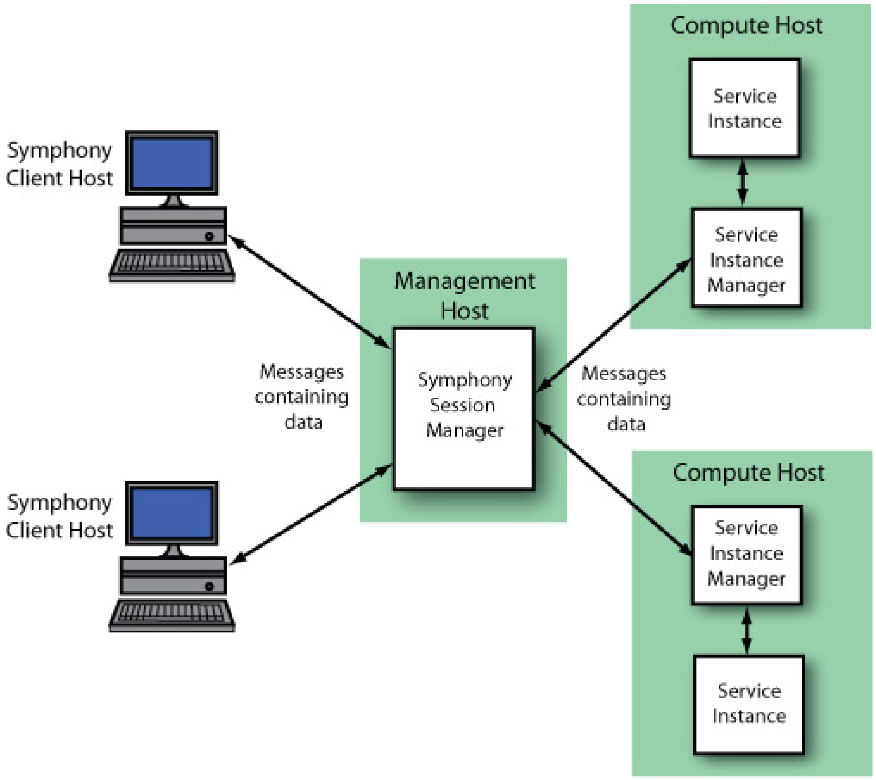
This section describes how data is submitted to a service using direct data transfer. But first, it is helpful to know how data is submitted in Symphony’s default model.

Default behavior without direct data transfer

Here is the sequence of events when a client wants to send task data to the service:

1. Client sends a task input message containing the data to the API layer, which serializes the message and submits it to the underlying communication layer.
2. The message is transferred by the communication layer to the session manager on the management host. The session manager replies to the client with an acknowledgement upon successful receipt of the message.
3. The session manager routes the message to the service instance manager and service instance on the compute host.
4. The service performs calculations on the input data within the message and returns the result to the client via the service instance manager and session manager.

The following diagram shows the data flow between the client and the service instance in Symphony's default model.



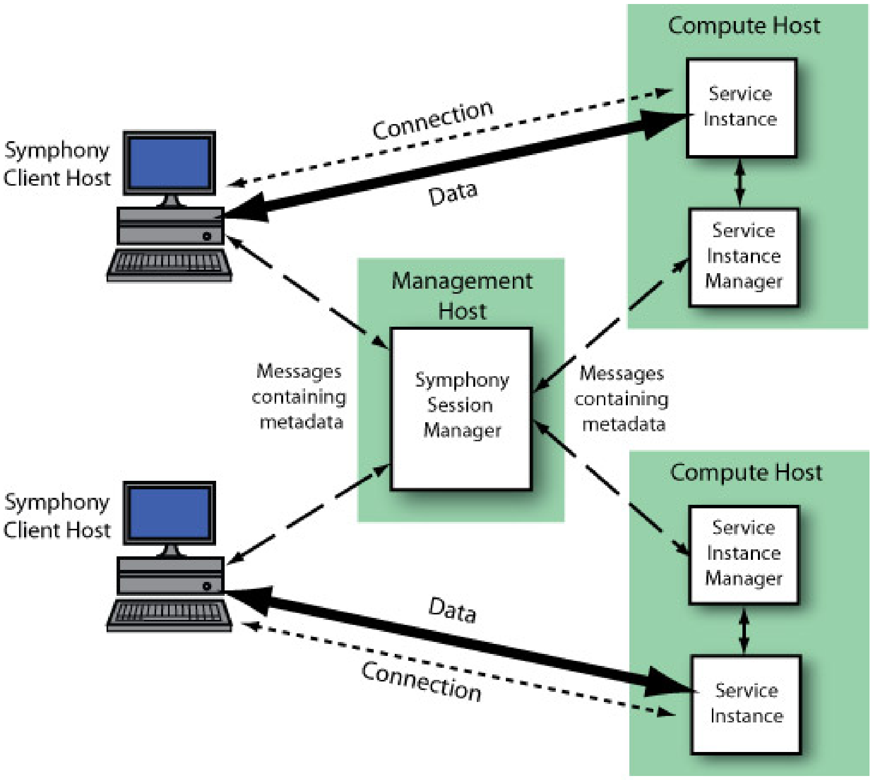
Behavior with direct data transfer

When direct data transfer is enabled for task input messages, the messages are sent to the service in the same manner as in Symphony's default model. The difference is when direct data transfer is enabled, the application data is not included in the task input message itself. Only metadata is actually sent with the task input message. To better understand the data flow between the client and the service when using direct data transfer for input/output messages, let's look at the sequence of events.

1. The client formulates a task input message encoded with the URL of the client (metadata). This is the URL the client will listen on.
2. The message is propagated to the service in the same manner as in Symphony's default model.
3. The service driver extracts the client URL and uses it to retrieve the data from the client.

4. The service performs calculations on the data and sends the resulting data directly to the client.
5. The client waits for acknowledgement from the session manager about the success of the task before accessing the output data locally.

The following diagram shows the data flow between the client and the service instance with direct data transfer enabled for input/output messages.



When to use direct data transfer

The direct transfer of application data should be considered in either of the following situations:

- You have many client connections being routed through a session manager. The session manager's routing and scheduling overhead can potentially impede data flow to the service.
- The client and service reside on the same subnet but the session manager does not.

Note:

Direct data transfer can be used in conjunction with data compression or other features such as common data updates. For example, if direct data transfer and data compression are both enabled, the compressed data will be sent directly to the service.

Client API

The direct data transfer feature can only be enabled through the client API at the session level.

Enabling direct data transfer for sessions

You can enable direct data transfer for all tasks associated with a session, and optionally, for common data and common data updates. To enable direct data transfer, the client application must do the following:

1. Create a session using the appropriate session attribute to inform the API of the client's intention to send data directly to the service. The session attribute is a member of the SessionCreationAttributes and SessonOpenAttributes classes.
2. Send the task input messages to Symphony.

The following code sample shows how direct data transfer is enabled using the SessionCreationsAttribute class in each supported language. For more information about the SessionCreationAttributes and SessionOpenAttributes classes, refer to the API reference documentation.

C++

```
SesssionCreationAttributes attributes;
attributes.enableDirectDataTransfer(true);
```

Java

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.enableDirectDataTransfer(true);
```

C# (.NET)

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.EnableDirectDataTransfer = true;
```

Setting direct data transfer flags

The direct data transfer flags allow greater control over Symphony behavior when the direct data transfer feature is enabled. By default, when direct data transfer is enabled, only the task data is sent directly between the client and service. This means that common data and common data updates are still sent to the service via the session manager. To override this behavior, it is necessary to set the appropriate direct data transfer flag.

For example, to set the direct data transfer flag for all tasks including common data and common data updates in a new session, the client application must do the following:

1. Create a session using the appropriate session attribute to inform the API to include common data and common data updates in the direct data transfer. The session attribute is a member of the SessionCreationAttributes class.
2. Send the task input messages to Symphony.

The following code sample shows how a direct data transfer flag is set with a SessionCreationAttributes object in each supported language. For more information about the SessionCreationAttributes class, refer to the API reference documentation.

C++

```
SessionCreationAttributes attributes;
attributes.setDirectDataTransferFlags(Session::IncludeCommonDataAndUpdates);
```

Java

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
```

```
attributes.setDirectDataTransferFlags
(DirectDataTransferFlags.INCLUDE_COMMON_DATA_AND_UPDATES);
```

C# (.NET)

```
SessionCreationAttributes attributes = new SessionCreationAttributes();
attributes.DirectDataTransferFlags =
DirectDataTransferFlags.IncludeCommonDataAndUpdates;
```

Port configuration

You can define a port or port range for the client to listen for connections from the service. You may want to do this if your client is running behind a firewall. The SOAM_DIRECT_DATA_PORT environment variable is used to define the port or port range; e.g., SOAM_DIRECT_DATA_PORT="25000" or SOAM_DIRECT_DATA_PORT="25000-25100".

Note:

If the SOAM_DIRECT_DATA_PORT is not defined, Symphony will use the value defined in EGO_CLIENT_ADDR. If neither of these variables are defined, Symphony randomly selects a client port to listen on.

Client memory management

Since direct data transfer will most likely be used in situations where an application needs to transfer large amounts of data, the client’s memory usage can become an issue. To conserve memory, Symphony can write the cached input/output data to disk and restore the data to memory only when it is required. Once the session is completed, Symphony removes all the input/output data from the file system.

For clients that have access to larger address space, for example, 64-bit clients, they have the ability to be optimized by keeping all the data in memory and relying on O/S paging instead of file caching.

The SOAM_DIRECT_DATA_STORAGE environment variable with possible values of *StoreInMemory* or *StoreOnDisk* is used to define whether the data resides in client memory or is written to disk. The default behavior when this variable is not defined is *StoreOnDisk*.

Multiple network interfaces

Symphony allows a non-default interface on the client host to be specified for communication with the service instance. The SOAM_DIRECT_DATA_ARRESSS environment variable can be defined with a valid IP address (or hostname alias) that represents the non-default interface.

Common data: using data in your application

Data: the issue

The data issue is a generic issue for financial applications. Regardless of how much data can be shared and pre-loaded in the service instances, you still need to pass some task-specific data between the Symphony client and service through the Symphony task message.

In many financial applications, a market dataset is often shared by more than one task. If you send this market dataset through task-specific input messages, the same data is sent more than once across the network. To reduce network traffic, maximize the shared data and minimize the task-specific data.

Common data

If you have input that is common to all tasks within a session, such as data that is required by every service invocation for a given session, you can create a `Message` object to encapsulate this common data. In this context, the `Message` object is referred to as *common data*.

Common data represents state that can be made available to service instances for the duration of a session.

When to use common data

Use common data when you need to set up the session-specific state of a service, and you only want to do it once per session, not on every task. Common data is useful for passing data from a client to a service. The service loads the data when the session is created.

You can use common data, for example, to set the environment in the service that is common to all tasks in a session. This way you only need to set the environment once, when the session is created.

Session-to-service instance affinity

Symphony attempts to use the same service instance for all tasks in a session as much as possible.

A service instance is made available to other sessions only when session workload completes, a session is closed or aborted, or when another more deserving session is assigned the service instance.

Lifetime of common data

You can access the common data in the service during execution of `onSessionEnter( )` on the ServiceContainer.

The common data object sent from the client can be deleted after the `Connection.CreateSession( )` call returns.

Common data update

Common data update allows a client application to send updates to existing common data after a session is created. You can send the updates from the client to your service by calling the update() method on the session object. You can access the updated common data in the service, during execution of `onSessionUpdate( )` on the ServiceContainer.

Client recovery

Disconnect and reconnect to a session

It is possible to disconnect from a session and reconnect.

You would want to do this when:

- You are creating a client that submits workload but does not have to wait for the results. Output can be retrieved anytime by other clients.
- A client abruptly disconnects from a session but can recover and wants to reconnect later.

Note:

When opening a session, to ensure that the client gets results in the expected manner:

- If you set the FetchResultsDirectly flag when creating a session, you should also set it when opening the session.
- If you did not set the FetchResultsDirectly flag when creating the session, you should not set it when opening the session.

About client disconnection

- There is no limit on the number of times you can connect to or disconnect from a session.
- Once a session is open on a connection, the connection has exclusive access to the session. Clients cannot simultaneously connect to a session. If a client attempts to connect to an existing session while another client has that session open, the client that was connected loses connection to the session and the new client gains access to the session.
- Clients which have successfully connected to existing sessions may get output that has already been delivered to a previous client. Client applications must be prepared to handle the same output delivered more than once.

Explicitly disconnect and reconnect to a session

You have a client that submits workload but does not have to wait for the results. Output can be retrieved anytime by other clients.

You can find sample code in the Session Reconnection sample in your Developer Edition installation

1. Disconnect from a session to be able to reconnect.
   a) Create a connection.
   b) Create a session.
   c) Store the session ID for later retrieval.
   d) Submit tasks.
   e) Close the session with the detach flag.

      This indicates to the session manager that workload should keep running and the session is to remain open.
2. Reconnect to the session
   a) In the same client or with a different client, create a connection.
   b) Open the session using the original session ID.
   c) Retrieve output and send more tasks as usual.

3. Allow output to be redistributable to a new client

   Optional.

   In some cases, you may want a new client to retrieve all the output. By default, session manager discards task results once output has been retrieved. You can configure the session type so that session manager will keep the output until the session is closed or aborted.

   a) Open your application profile.
   b) In the SessionType, add the parameter discardResultsOnDelivery=false.

```
<SessionTypes>
        <Type name="DefaultSession" priority="1"recoverable="false"
abortSessionIfClientDisconnect="true" sessionRetryLimit="3" taskRetryLimit="3"
abortSessionIfTaskFail="false" suspendGracePeriod="100" taskCleanupPeriod="250"
persistSessionHistory="all" persistTaskHistory="error" discardResultsOnDelivery="false"/>
</SessionTypes>
```

   c) Save your application profile.
   d) Update your application profile with the `soamreg` command.

Reconnect to a session after client recovery

You have a client that terminated abnormally in the middle of a long-running session. The client may be able to recover and continue working without losing its workload.

1. Store the session ID.

   When creating a session in your client, store the session ID for later retrieval. This is important. If your client terminates abnormally, you will be able to reconnect to the session.
2. Configure your session type to not clean up workload when the client disconnects.

   By default, once a client disconnects, workload is cleaned up by the session manager. If your client is recoverable and will attempt to reconnect to sessions, configure workload to not be cleaned up.

   a) Open your application profile.
   b) In the SessionType, set the parameters abortSessionIfClientDisconnect=false and discardResultsOnDelivery=false.

```
 <SessionTypes>
        <Type name="DefaultSession" priority="1"
recoverable="true"abortSessionIfClientDisconnect="false" sessionRetryLimit="3" taskRetryLimit="3"
abortSessionIfTaskFail="false" suspendGracePeriod="100"  taskCleanupPeriod="250"
persistSessionHistory="all" persistTaskHistory="error" discardResultsOnDelivery="false"/>
</SessionTypes>
```

   c) Save your application profile.
   d) Update your application profile with the `soamreg` command.
3. Reconnect when your client recovers.

   When your client terminates abnormally, restart it.

   a) Restart your client when it terminates abnormally.
   b) Create connection.
   c) Open the session using the original session ID.
   d) Continue to operate with the session as usual.

      You can retrieve output or send more tasks as usual.

Remote clients

Configuration of TCP connections

This topic is only applicable on Symphony grid.

TCP connection attributes are configured on a cluster basis to optimize data throughput over network connections. These attributes are set in the `sd.xml` configuration file.

The attributes should be configured for each connection endpoint in the Symphony environment, i.e., the client, Session Director, and session manager. The following table lists the relevant attributes.

Tip:

If you want to configure attributes with default values, it is not necessary to add them to the sd.xml file.

Attribute	Default	Notes
TCP_NODELAY	1	May be set to 0 for message aggregation.
TCP_KEEP_ALIVE_TIME	Value derived from current OS setting	OS default is 7200 seconds for most operating systems but may vary.
TCP_SEND_BUFFER_SIZE	65535	Any new value that is less than or equal to the default value is ignored.
TCP_RECV_BUFFER_SIZE	65535	Any new value that is less than or equal to the default value is ignored.

Note:

On Solaris platforms, TCP_KEEP_ALIVE_TIME can only be set system-wide and not on a per socket basis (this is an OS limitation). Symphony on Solaris ignores the TCP_KEEP_ALIVE_TIME option if it is set.

Sample configuration in sd.xml:

```
<ego:EnvironmentVariable name="SDK_TRANSPORT_OPT">TCP_NODELAY=0,TCP_KEEP_ALIVE_TIME=300,
TCP_SEND_BUFFER_SIZE=65536,TCP_RECV_BUFFER_SIZE=65536</ego:EnvironmentVariable>
         ...
<ego:EnvironmentVariable name="SD_SDK_TRANSPORT_OPT">TCP_NODELAY=0,TCP_KEEP_ALIVE_TIME=300,
TCP_SEND_BUFFER_SIZE=65536,TCP_RECV_BUFFER_SIZE=65536</ego:EnvironmentVariable>
         ...
<ego:EnvironmentVariable name="SSM_SDK_TRANSPORT_OPT">TCP_NODELAY=0,TCP_KEEP_ALIVE_TIME=300,
TCP_SEND_BUFFER_SIZE=65536,TCP_RECV_BUFFER_SIZE=65536</ego:EnvironmentVariable>
         ...
```

Configuring local connections on clients

There may be situations where global TCP connection attributes are not appropriate for all connection endpoints in the system; for example, remote clients that are geographically distant from the cluster may require more time to send messages over the network. It is possible to override the system-wide attributes on the remote client host by setting environment variables in the OS shell before starting the client process. This method of overriding system-wide attributes can also be applied to remote hosts that are running services.

The environment variables correspond to the four TCP connection attributes described previously:

Environment variable on client	Overrides this attribute in SD.xml
PLATCOMMDRV_TCP_NODELAY	TCP_NODELAY
PLATCOMMDRV_TCP_KEEPALIVE_TIME	TCP_KEEP_ALIVE_TIME
PLATCOMMDRV_TCP_SEND_BUFFER_SIZE	TCP_SEND_BUFFER_SIZE
PLATCOMMDRV_TCP_RECV_BUFFER_SIZE	TCP_RECV_BUFFER_SIZE

Once the environment variables have been created, you can adjust their values to suit the network environment.

Connect to different clusters with the same client

- This section applies to Symphony grid, but not Symphony DE.
- There is no firewall that prohibits the client from establishing communication ports to management hosts in the clusters.
- All clusters to which the client is connecting have the same Symphony version and same patch level .
- All cluster configuration files have the same security plugin configured. The default cluster configuration file `%EGO_CONFDIR%\ego.conf` on Windows and `$EGO_CONFDIR/ego.conf` on Linux.

You want a client to connect to the same application in a different cluster and to be able to submit work there. This is useful when you have several clusters and you want to take advantage of idle CPUs.

1. In your client code, when creating a connection, use the connection method that allows you to specify a connection file name or master host list.
   - Use a file name. For example:
     - Windows—`file://c:\ego\kernel\conf\ego.conf`
     - Linux—`file:///opt/ego/kernel/conf/ego.conf`
   - Use a cluster URL. For example:

     ```
     master_list://host1:7870 host2:7870 host3:7870
     ```

     When you specify a cluster URL, you use the keyword master_list, and indicate the master host name and port number.

     Master hosts and master candidate hosts are specified in the `ego.conf` file in the cluster installation with the parameter EGO_MASTER_LIST.

     The port number to use for connection is identified with the EGO_KD_PORT parameter in the cluster's `ego.conf` file. All ports numbers must be the same.

Test your connection to multiple clusters. Note that code for this feature can be tested in Symphony DE, but you will only be able to connect to multiple clusters once in the grid environment.

Web Service clients

Web Services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. This section begins with a description of the major components and concepts of a Web Service. Later, we describe these concepts within the Symphony environment.

For more information about developing a Symphony Web Service client, refer to the Admin Web Service client tutorial in the Knowledge Center.

Web Service components

XML

XML is used in the Web Services architecture as the platform-independent format for transferring information between the Web Service and the Web Service client. The XML format ensures uniform data representation and exchange.

WSDL

The Web Services Description Language (WSDL) describes the message syntax associated with the invocation and response of a Web Service. A WSDL file is an XML document that defines the Web Service operations and associated input/output parameters. In a way, the WSDL can be considered a contract between the Web Services client and the Web Services server.

Basically, a WSDL document describes three fundamental properties of a Web Service:

- The operations (methods) that the service provides including input arguments needed to invoke them and the response.
- Details of the data formats and protocols required to access the service’s operations.
- Service location details such as a URL.

WSDL 1.1 was suggested in a note to W3C as an XML format for describing Web Services; refer to http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

XML Schema

XML schemas are used to specify the structure of WSDL documents and the data type of each element/attribute. XML schemas describe the documents that serve as the body of the SOAP messages traversing the Symphony DE Web Service interface.

SOAP

SOAP is the protocol used for communication between the Web Service and the client application. SOAP uses the Hypertext Transfer Protocol (HTTP or HTTPS) as the underlying protocol for transporting the data. SOAP 1.1 was suggested in a note to W3C as a protocol for exchanging information in a distributed environment. Refer to http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

Web Service security

The Symphony Web Services implementation supports the use of UserNameToken-based authentication between the Web Service client and the Symphony Web Service. Refer to UsernameToken Profile 1.0 (OASIS Web Security Standard 200401) for further information.

A closer look at a Symphony WSDL and schema

This section looks at some key features of a Symphony WSDL and schema.

SOAP binding style

SOAP supports two invocation models: Remote Procedure Calls (RPC) and document style.

In the RPC-style model, clients invoke the Web Service by sending parameters and receiving return values that are wrapped inside the SOAP body. These procedure calls are synchronous, which means that the client sends the request and waits for the response.

In the document style model, the client sends the parameters to the Web Service within an XML document. The Web Service receives the entire document, processes it and possibly returns a response message. Using the document style, the body of the SOAP message is interpreted as straight XML. Hence, this combination of sending a document with a literal XML infoset as a payload is referred to as document/literal. This is opposed to the RPC style that uses RPC conventions for the SOAP body as defined in the SOAP specification. One advantage of using the document-centric approach is that document messaging rules are more flexible than the RPC style, which allows for changes to the XML schema without breaking the calling applications.

The type of binding model that is implemented is determined by an attribute in the WSDL. The style attribute within the SOAP protocol binding can be set to either RPC or document. Symphony WSDLs use document style binding and that is why a document must be created for the request and response messages. Here is an example of a WSDL binding element that is set to document style. Note that the encoding technique is specified by the soap:body element's use attribute. In this case, it is set to literal.

```
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
...
 <input>     <soap:body use="literal"/> </input>
```

Passing parameters to a Web Service

The following example shows portions of the WSDL file for the Symphony DE Admin Web Service.

```
...
<types><schema targetNamespace="http://www.platform.com/soam/v2/wsse.xsd"
...
 <element name="sdViewSession">
   <complexType>
    <sequence>
     <element name="appName" type="xsd:string" minOccurs="0" maxOccurs="1"
     nillable="true"/>
     <element name="sessionId" type="soam:SessionID" minOccurs="1" maxOccurs="1"/>
     <element name="filter" type="xsd:string" minOccurs="0" maxOccurs="1"
     nillable="true"/>
     <element name="maxCap" type="xsd:long" minOccurs="1" maxOccurs="1"/>
    </sequence>
   </complexType>
 </element>
...
<message name="sdViewAppResponse"> <part name="parameters" element="soam:sdViewAppResponse"/
></message><message name="sdViewSession"> <part name="parameters"
element="soam:sdViewSession"/></message>
...
<portType name="SoamPortType">
...
<operation name="sdViewSession">
  <documentation>Service definition of function soam__sdViewSession</documentation>
  <input message="tns:sdViewSession"/>
  <output message="tns:sdViewSessionResponse"/>
 </operation>
...
```

In this example, we look at the sdViewSession operation. This operation takes a single input argument defined as a message of type sdViewSession. (In the WSDL context, all parameters are called messages.) Next, we determine the number of parameters in this message and their data types.

```
...
<portType name="SoamPortType">
...
 <operation name="sdViewSession">
  <documentation>Service definition of function soam__sdViewSession</documentation>
  <input message="tns:sdViewSession"/>
  <output message="tns:sdViewSessionResponse"/>
 </operation>
...
```

If you look up the sdViewSession type in the types element, you will find a complex data type containing four elements (parameters): appName, sessionId, filter, and maxCap. These parameters have string, sessionId, string, and long data types, respectively. When you call the sdViewSession operation, you pass all four parameters as input.

```
<message name="sdViewSession">
 <part name="parameters" element="soam:sdViewSession"/>
</message>
...
<element name="sdViewSession">
 <complexType>
  <sequence>
   <element name="appName" type="xsd:string" minOccurs="0" maxOccurs="1"
   nillable="true"/>
   <element name="sessionId" type="soam:SessionID" minOccurs="1" maxOccurs="1"/>
   <element name="filter" type="xsd:string" minOccurs="0" maxOccurs="1"
   nillable="true"/>
   <element name="maxCap" type="xsd:long" minOccurs="1" maxOccurs="1"/>
  </sequence>
 </complexType>
</element>
...
```

Return values from a Web Service

Web Service operations often return information back to the client application. You can determine the name and data type of returned information by examining the WSDL or schema files for the Web Service.

Referring to the previous portion of the WSDL file for the Admin Web Service, we find that the operation named sdViewSession returns a message of type sdViewSessionResponse. If you look up the sdViewSessionResponse type in the types element, you will find it contains a return argument called sessionAttrVector. You can see that the sdViewSessionResponse element has a complex data type. Complex data types are serialized as XML and returned from the Web Service as the result. The variable used to store the result must match the structure of the complex data type.

```
<element name="sdViewSessionResponse">
 <complexType>
  <sequence>
   <element name="sessionAttrVector" type="soam:SessionAttributeVector" minOccurs="1"
   maxOccurs="1" nillable="false"/>
  </sequence>
 </complexType>
</element>
```

Building a Web Service client

A Web Services client is an application capable of sending and receiving SOAP messages. Such an application serializes or deserializes the SOAP messages to a programming language type system enabling programmatic processing.

Here is the sequence for invoking a Web Service:

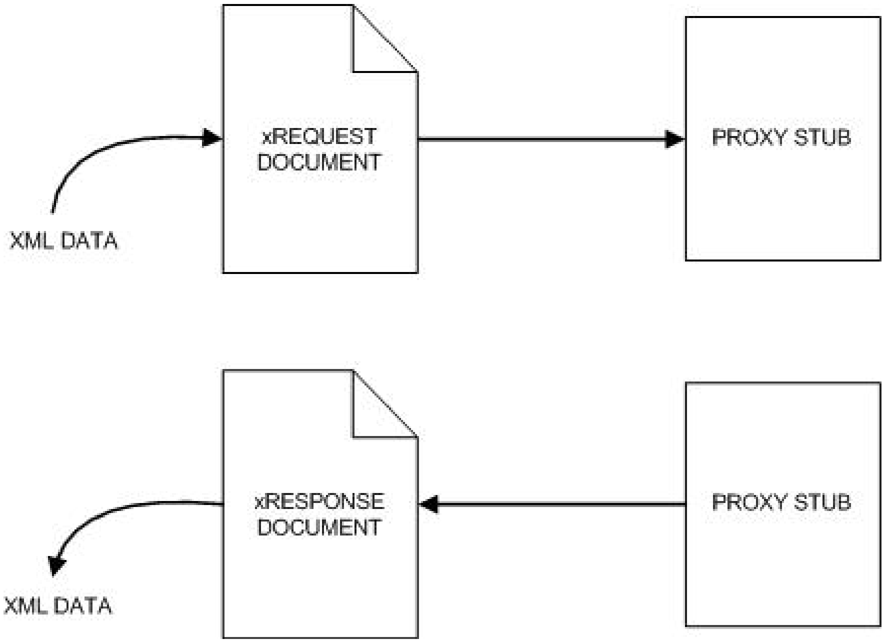
- Client serializes the arguments of the method call into the XML payload of the SOAP message
- Send the message to the Web Service
- Wait for a response (or timeout)
- Deserialize the XML payload in the response message to a local type/structure
- Return that type/structure as a value from the method call.

Using Axis2 to Develop Java Web Service Clients

As a client to a Web Service, encoding your requests in XML to the Web Service and decoding the responses you get back would be tedious (not to mention implementing the logic that deals with accepting requests and sending responses).

Apache Axis2 is an implementation of the SOAP protocol and it shields the developer from the details of dealing with SOAP and WSDL. You can use Axis on the client side to greatly facilitate the development of your client. Bear in mind that there are several tools available to aid in the development of a Web Service client and Platform does not endorse any particular one.

When using Axis2 to write your client, you don't need to deal directly with SOAP and XML. Axis creates a proxy (or stub) for your clients to abstract away SOAP. All you need to do is make the method calls on the Web Service proxy as if it were a local object.



The client calls the stub, the stub translates the call into a SOAP message, and the stub sends it to the Web Service. The listening server receives the SOAP message and translates it into a method call at the server. Since the server is written in Java, the SOAP message is turned into a Java call. The server's return values are translated back to SOAP and then returned to the stub, which translates the returned SOAP message into a response to the client.

A sample Bash shell script is provided that creates client-side classes for consuming services described in the WSDL file.

```
#!/bin/bash
# Add the location of Java tools to PATHexport PATH=/usr/local/jdk/bin/:$PATH
# Set the location of Axis2 binary installation
AXIS2_HOME=/home/ACCOUNT DIRECTORY/axis2-0.92-bin
AXIS2_LIB=../ego/axis2
# Build Axis2 classpath
AXIS2_CLASSPATH=.
for i in $AXIS2_HOME/lib/*.jar; do AXIS2_CLASSPATH=$AXIS2_CLASSPATH:$i;
done
SCHEMAS="Soam.wsdl"
#Cleanup
rm -fr ./src ./respources *.jar
cp ../../../Soam.wsdl .
# Generate the Java classes
for i in $SCHEMAS; do echo $i; j=`echo $i | sed -e 's/\(.*\)\..*/\1/'`; echo $j ;
java -classpath $AXIS2_CLASSPATH org.apache.axis2.wsdl.WSDL2Java -uri $i -o src
done
# Compile the generated classes
cd src
for i in codegen codegen/databinding/com/platform/www
 codegen/databinding/com/platform/www/impl codegen/databinding/org/w3/www codegen/databinding/
org/w3/wwwimpl codegen/databinding/org/xmlsoap/schemas codegen/databinding/org/xmlsoap/
schemas/impl;
do
javac -classpath $AXIS2_CLASSPATH $i/*.java;
done
# Create the Jar file
jar cf soamAdmin.jar ./codegen ./schemaorg_apache_xmlbeans/
mv soamAdmin.jar ..
#Cleanup
cd ..
#rm -fr ./src
# Use the generated jar file in classpath of your application
exit 0
```

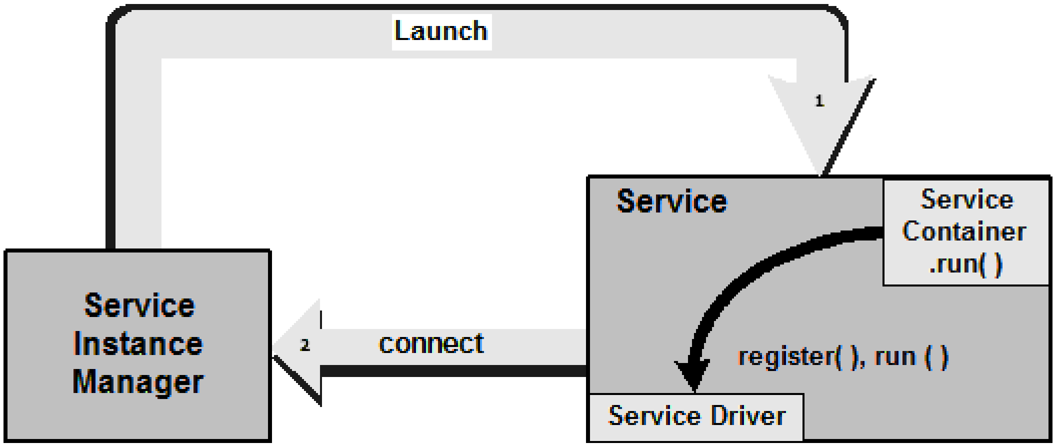
CHAPTER

4

Developing Services

About services

A service is a process on its own: anything that is executable and can access the Symphony API can implement a service. The service does not initiate any actions on its own. Symphony triggers all service actions.



Service containers and service instances

You create your service by extending the `ServiceContainer` class and implementing the required handler methods.

A running instance of a `ServiceContainer` is called a service instance.

Symphony assigns a number of service instances to a session according to policy, session priority, and the number of tasks in the session.

A service instance can be used to service different sessions within the same application. That is, Symphony does not need to destroy an existing service instance and start a new service instance to serve different sessions within the same application.

Service structure

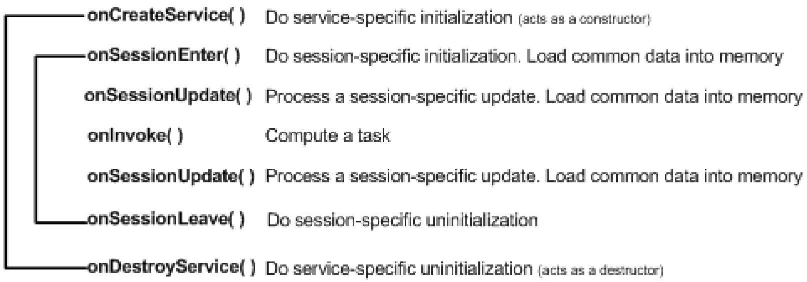
The `onInvoke ( )` method in `ServiceContainer( )` is the only required method. All other methods are optional.

Do not do anything in your service until you:

1. Create a service container
2. Call run on the service container

When the application calls the ServiceContainer run( ) method, the service instance process registers with Symphony. The service instance is given a certain time to register before it is considered to have timed out. If your application does lengthy operations before calling the ServiceContainer run( ) method, your service instance may not be able to register before the time out expires. This is considered a failure to start the service instance.

The following diagram indicates the structure of a service with all methods implemented, in the order that they need to be implemented. Note method pairs: if you use `onCreateService ()`, you must also use `onDestroyService()`.


onCreateService( ) Do service-specific initialization (acts as a constructor)
onSessionEnter( ) Do session-specific initialization. Load common data into memory
onSessionUpdate( ) Process a session-specific update. Load common data into memory
onInvoke( ) Compute a task
onSessionUpdate( ) Process a session-specific update. Load common data into memory
onSessionLeave( ) Do session-specific uninitialization
onDestroyService( ) Do service-specific uninitialization (acts as a destructor)

API method	Description	Action system takes
onCreateService( )	Optional. Use this method to perform any required environment set up when your service is initialized.	Symphony invokes `onCreateService( )` just after it launches the service instance to initialize the service instance. This method is called once per service instance that is started.
onDestroyService()	Optional. Use this method to clean up a service instance before it is destroyed.	Symphony invokes `onDestroyService( )` to clean up the service instance when a service instance is ending its lifecycle, provided that `onCreateService( )` has been called and has returned without exception. **Note:** This method will not be called if a service instance crashes or exits or if a timeout occurs.
onSessionEnter( )	Optional. Use this method to load session common data into memory.	Symphony invokes `onSessionEnter( )` when a service instance is assigned to a session. **Note:** Note that if there is no common data in createSession( ), onSessionEnter( ) is not called.
onSessionUpdate( )	Optional. Use this method to load session updated common data into memory.	Symphony invokes `onSessionUpdate( )` when a session-specific update has been propagated to the service instance. **Note:** Note that if there is no common data in createSession( ), onSessionUpdate( ) is not called.

API method	Description	Action system takes
onSessionLeave()	Optional. Use this method to free up session common data that was loaded into memory with onSessionEnter().	Symphony invokes `onSessionLeave()` to do session-specific uninitialization when the service instance is unassigned from the session, provided that `onSessionEnter()` has been called and has returned without exception. **Note:** This method will not be called if a service instance crashes or exits, a timeout occurs, or there is no common data in createSession( ).
onInvoke( )	Required. Use this method to compute a task.	Symphony invokes `onInvoke()` to compute a task. This method is called once for every input message that is sent to the service instance. The method can be called in the same service instance multiple times to compute multiple tasks for the same session or for different sessions.

Service main( )

The service main() is the entry point to the service. Create the container and run it.

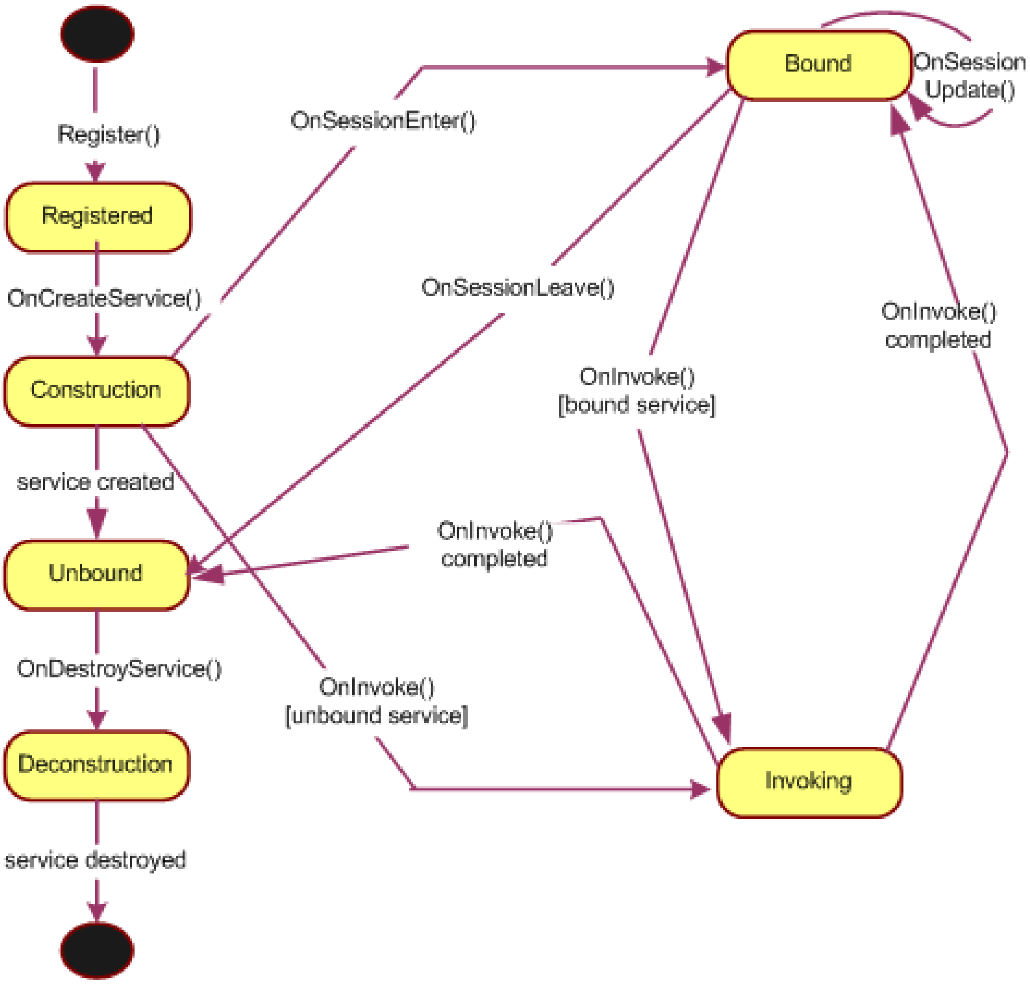
Restrictions

- Do not implement any service initialization before calling the ServiceContainer::run() method. If any service initialization needs to be done, implement the onCreateService() handler for your service container.
- Do not implement any service uninitialization after calling the ServiceContainer::run() method. If any service uninitialization needs to be done, implement the onDestroyService () handler for your service container since there is no guarantee that the remaining code in main() will execute after calling ServiceContainer::run().

Service lifecycle

Symphony triggers state changes for the `ServiceContainer` as illustrated in the following diagram. The calls indicated on the diagram are calls made in the service, with the exception of Register() which is an internal call made by the system.

The arrows indicate a normal return of the method.



Note:

`OnSessionEnter( )` and `OnSessionLeave( )` are not called if the client does not relay common data when creating a session. `OnSessionUpdate( )` is not called if the client does not relay common data updates to the session.

Service instance lifecycle

A service instance is an executing instance of a service. There can be many instances of a service at any one time. Service instances are created by service instance managers.

Service instances can be started either before or when they are assigned to a session. They can stay running to compute multiple tasks of the same session to use the data and state information cached in memory for better performance. They can either exit or continue running when their serviced session is finished.

A service is transient if the service instances start and exit per session. A service is persistent if the service instances stay and serve multiple sessions. A persistent service is a long-running process like a daemon, which has to be more carefully programmed to avoid any accumulated problems such as memory leaks.

By default, the service instances in Symphony persist for multiple sessions. To make a service instance transient, you can return a control code from `onSessionLeave()` to tell Symphony to restart the service instance once it leaves the current session. This can also be used when you want to clean up any accumulated problems by restarting your service from time to time.

Timeouts that affect service instance life cycle

The service instance lifecycle can be affected by different configured timeouts. If an application has timeouts configured, then Symphony will take action if an operation exceeds the configured timeout. In this case, Symphony terminates the service instance, causing the cleanup methods not to execute under the following circumstances:

Method	**Not called**
onDestroyService()	• When an invocation of one of the following service methods times out: ◦ SessionEnter ◦ SessionUpdate ◦ Invoke ◦ SessionLeave • When a task cannot complete before the suspendGracePeriod expires. • When a task cannot complete before the taskCleanupPeriod expires. • When a resource on which the service instance is running is reclaimed, and the service instance cannot clean up before the applied reclaim grace period expires. • When the application is disabled or unregistered and the service instance cannot clean up before the cleanupTimeout expires. • When a middleware component becomes unavailable and the service instance cannot clean up before the cleanupTimeout expires.
onSessionLeave()	• When an invocation of one of the following service methods times out: ◦ SessionUpdate ◦ Invoke • When a task cannot complete before the suspendGracePeriod expires. • When a task cannot complete before the taskCleanupPeriod expires. • When a resource on which the service instance is running is reclaimed, and the service instance cannot clean up before the applied reclaim grace period expires. • When the application is disabled or unregistered and the service instance cannot clean up before the cleanupTimeout expires. • When a middleware component becomes unavailable and the service instance cannot clean up before the cleanupTimeout expires.

Feature: Access to application attributes in a service

The service API gives developers access to information exposed to the Service Instance.

Scope

Applicability	Details
Operating system	• Windows • Linux • Solaris
Limitations	None

About application attributes for services

The service API enables developers to access the following application attributes from the ServiceContext object:

- Application name
- Service name
- Consumer ID
- Deployment directory
- Configured log directory

Example: Get application attributes with service API

To retrieve and print all the additional information your service code would look like this:

C++

```
const char* applicationName = serviceContext.getApplicationName();
const char* consumerId = serviceContext.getConsumerId();
const char* deploymentDir = serviceContext.getDeployDirectory();
const char* logDir = serviceContext.getLogDirectory();
const char* serviceName = serviceContext.getServiceName();
...
```

C# (.NET)

```
string applicationName = serviceContext.ApplicationName;
string consumerId = serviceContext.ConsumerId;
string deploymentDir = serviceContext.DeployDirectory;
string logDir = serviceContext.LogDirectory;
string serviceName = serviceContext.ServiceName;
...
```

Java

```
String applicationName = serviceContext.getApplicationName();
String consumerId = serviceContext.getConsumerId();
String deploymentDir = serviceContext.getDeployDirectory();
String logDir = serviceContext.getLogDirectory();
String serviceName = serviceContext.getServiceName();
...
```

Error codes and embedded service exceptions

When an exception is thrown from within the service of an application during its invocation, during the processing of common data when the invocation of OnSessionEnter causes the session to be aborted, or during the processing of common data update when the invocation of OnSessionUpdate causes the session to be aborted, that exception is propagated to the client. The exception propagated to the client describes the error as it relates to the middleware. This error may or may not be immediately useful to the application since it presents itself in a format that is external to the application.

It is possible for an application to get more specific details about the error returned. The application must attach enough meaningful information to the exception before it is thrown within the service. In general, you can attach application-specific error information like a string description and/or an error code to an exception before throwing it in the service code. The error information is preserved and wrapped within a larger Symphony exception, which is then propagated to the client.

This information can then be used to further qualify how the client responds to any failures it receives through an exception. For example, when a task fails, you may need to be able to determine whether an error pertains to the service or to the Symphony system itself.

Attach an error code to service exceptions

To associate an error code to service exceptions, when creating the error message as a FatalException or FailureException, add the error code and/or description to the exception when it being constructed.

Retrieve embedded service exceptions

Service-specific exceptions thrown within the context of OnInvoke(), OnSessionEnter(), or OnSessionUpdate() are literally preserved and can be retrieved from the Symphony exception by the client. Note that stack trace preservation is not available. To retrieve the service-specific exception, use the SoamException::getEmbeddedException() method. Note that you must check for null before attempting to use the embedded exception. This embedded exception contains any specific information that was added in the service side of the application before the exception was thrown. If there is no exception embedded, then it is likely that the exception is being reported from the system, or the exception occurred as a result of an unexpected failure in the service code itself.

Example

The following pseudo-code illustrates how to retrieve embedded exceptions and associate error codes with exceptions. This example relates to a task failing and how it is handled.

```
If task was successful
{
Process task normally
}
else
{
It must have failed so handle the failure case
// get the exception associated with this task
SoamException ex = output.Exception;
Console.WriteLine(ex.ToString());
SoamException myEx = ex.EmbeddedException;
if (myEx != null)
 {
  if (myEx.ErrorCode == myErrorCodes.error1)
   {
```

```
   Handle specific application error
   ...
   }
  else
  if (myEx.ErrorCode == myErrorCodes.error2)
   {
   Handle specific error
   ...
   }
 }
}
```

Service error handling and host blocking

Feature: Service error handling control

With service error handling control, you configure events and corresponding actions to take on sessions, tasks, and service instances when a service is in a specific state of its lifecycle. This feature also enables you to configure timeouts for all methods within the service and actions to take when a timeout occurs.

For example, in the service onInvoke() call, you could configure that if the method exits, the system is to restart the service.

Scope

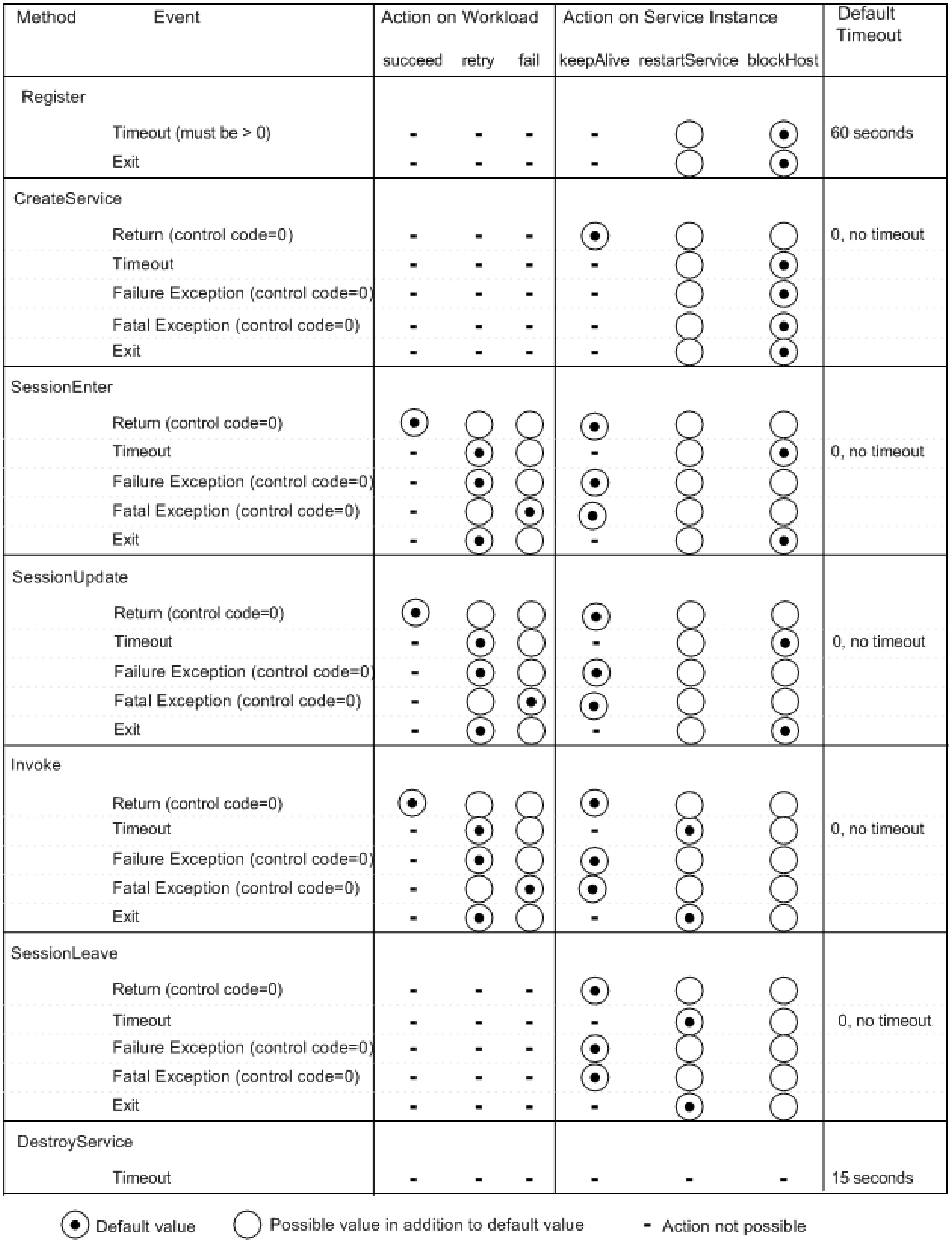
Applicability	Details
Operating system	• Linux, UNIX • Windows
Allows for	• Configuration of actions the system should take upon failure exceptions, fatal exceptions, exit, and return for service lifecycle methods: • Register() • onCreateService() • onSessionEnter() • onSessionUpdate() • onInvoke() • onSessionLeave() • onDestroyService() • Configuration of timeout values for service lifecycle methods and actions to take. • Configuration of custom control codes upon which the system is to take specific actions.
Dependencies	n/a
Limitations	For backwards compatibility, Symphony accepts Symphony 3.1 and later formats in the <Control></Control> section of the application profile. Application profiles that are registered with the 3.1 format are modified by the system to the current format. Note that you cannot use both Symphony 3.1 and a later format in the same application profile.

About service error handling control

Default and possible configurations for service error handling

The following table lists API methods that can be used in service code, possible events that can be configured for each method, and possible actions that can be taken on workload (session or task) and on service instances upon trigger of the event.

In the table, a default control code of 0 is assumed.



Method	Event	Action on Workload			Action on Service Instance			Default Timeout
		succeed	retry	fail	keepAlive	restartService	blockHost	
Register								
	Timeout (must be > 0)	-	-	-	-	○	⊙	60 seconds
	Exit	-	-	-	-	○	⊙	
CreateService								
	Return (control code=0)	-	-	-	⊙	○	○	0, no timeout
	Timeout	-	-	-	-	○	⊙	
	Failure Exception (control code=0)	-	-	-	-	○	⊙	
	Fatal Exception (control code=0)	-	-	-	-	○	⊙	
	Exit	-	-	-	-	○	⊙	
SessionEnter								
	Return (control code=0)	⊙	○	○	⊙	○	○	
	Timeout	-	⊙	○	-	○	⊙	0, no timeout
	Failure Exception (control code=0)	-	⊙	○	⊙	○	○	
	Fatal Exception (control code=0)	-	○	⊙	⊙	○	○	
	Exit	-	⊙	○	-	○	⊙	
SessionUpdate								
	Return (control code=0)	⊙	○	○	⊙	○	○	
	Timeout	-	⊙	○	-	○	⊙	0, no timeout
	Failure Exception (control code=0)	-	⊙	○	⊙	○	○	
	Fatal Exception (control code=0)	-	○	⊙	⊙	○	○	
	Exit	-	⊙	○	-	○	⊙	
Invoke								
	Return (control code=0)	⊙	○	○	⊙	○	○	
	Timeout	-	⊙	○	-	⊙	○	0, no timeout
	Failure Exception (control code=0)	-	⊙	○	⊙	○	○	
	Fatal Exception (control code=0)	-	○	⊙	⊙	○	○	
	Exit	-	⊙	○	-	⊙	○	
SessionLeave								
	Return (control code=0)	-	-	-	⊙	○	○	
	Timeout	-	-	-	-	⊙	○	0, no timeout
	Failure Exception (control code=0)	-	-	-	⊙	○	○	
	Fatal Exception (control code=0)	-	-	-	⊙	○	○	
	Exit	-	-	-	-	⊙	○	
DestroyService								
	Timeout	-	-	-	-	-	-	15 seconds

⊙ Default value ○ Possible value in addition to default value - Action not possible

Service methods for which you can define events and actions

The following table lists the service methods for which you can define events and actions.

Event	Description
Register()	Register() is an internal method used by the system.
onCreateService()	Create the service container.
onSessionEnter()	Get common data and store it for later with the session context.
onSessionUpdate()	Get an update to existing common data
onInvoke()	Process the input message.
onSessionLeave()	Free the common data and all updates to common data for the session. Used with onSessionEnter().
onDestroyService()	Destroy and unload the service instance from the service container. The service instance is no longer associated with the session.

Events that trigger actions on workload and service instances

Condition	Description
Return	Defines the action to take upon successful return of the method.
Timeout	Defines the action to take when the method times out. If you know that a service method invocation should not exceed a certain amount of time, you can configure it to be terminated after a specific time period has elapsed.
Failure Exception	Defines the action to take when a failure exception occurs within the specified method. A FailureException indicates that the operation failed in the service but is worth trying on a different compute host.
Fatal Exception	Defines the action to take when a fatal exception occurs within the specified method. A FatalException indicates that the operation failed in the service and is not likely to be successful if attempted on a different compute host.
Exit	Defines the action to take on the service instance when the service exits while invoking the method.

Actions that can be taken on workload

The following actions are possible on workload (sessions and tasks). Note that the possible combination of actions varies according to the method.

Possible action	Description
retry	When a specified event occurs, retry the method up to the number of times configured by the session and task retry limits in the application profile. For SessionEnter and SessionUpdate, the system attempts to bind the session to the service instance up to the sessionRetryLimit in the application profile before the session is aborted. **Note:** The retry count for both of these methods are considered together. For example, if SessionEnter fails once and SessionUpdate fails twice, then the session rerun count is equal to 3. Therefore the SessionRetryCount should be set to a value that accounts for both SessionEnter and SessionUpdate failures. For Invoke, the system attempts to run the task up to the taskRetryLimit defined in the application profile before the task is failed.
fail	When a specified event occurs, abort the session or fail the task, and propagate errors to the client application. For SessionEnter or SessionUpdate, immediately abort the session. Do not retry the method. For Invoke, immediately fail the task. Do not retry the method.
succeed	When the method succeeds, continue taking the action in the method until completion. No further action is taken on workload. This is a normal return.

Actions that can be taken on service instances

The following actions are possible on the service instance. Note that the possible combination of actions varies according to the method.

Possible action	Description
blockHost	When the specified event occurs, terminate the running service instance on this host and do not use this host to start any other service instance for the application. The host on which the service instance was running is added to the blocked host list for the application. This host is no longer selected to run work for the application until it is explicitly unblocked through the EGO command-line or the Platform Management Console.
restartService	When the specified condition occurs, terminate the service instance, start a new service instance on the same host, and recover state. There is no limit to the number of times that a service instance can be restarted.
keepAlive	When the specified condition occurs, take no action on the running service instance.

Control codes

The control code is an integer returned either normally through the return of the method, or returned when a fatal or failure exception occurs. You can configure specific numbers to trigger actions in all methods except Register() and DestroyService().

The default value for control codes is 0. So if you do not explicitly set a control code in your service, then return is considered to have a control code of 0. Symphony executes the behavior defined for control code ="0" for the service event that occurs.

If you want, for example, to sometimes restart the service instance and sometimes not, then use any number other than 0 for your control code. For example, if you want to restart the service instance on every 10th invoke, indicate a code of 1. A code of 0 indicates the default action.

Configuration to modify service error handling

You can modify service error handling behavior in the application profile, Control section.

Configuration format

```
<Control>
    <Method name="Register" >
       <Timeout duration="60" actionOnSI="blockHost"/>
        <Exit actionOnSI="blockHost"/>
    </Method>
    <Method name="CreateService" >
       <Timeout duration="0" actionOnSI="blockHost"/>
       <Exit actionOnSI="blockHost"/>
       <Return controlCode="0" actionOnSI="keepAlive"/>
       <Exception type="failure" controlCode="0" actionOnSI="blockHost"/>
       <Exception type="fatal" controlCode="0" actionOnSI="blockHost"/>
   </Method>
   <Method name="SessionEnter" >
       <Timeout duration="0" actionOnSI="blockHost" actionOnWorkload="retry"/>
       <Exit actionOnSI="blockHost" actionOnWorkload="retry"/>
       <Return controlCode="0" actionOnSI="keepAlive" actionOnWorkload="succeed"/>
       <Exception type="failure" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="retry"/>
      <Exception type="fatal" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="fail"/>
   </Method>
   <Method name="SessionUpdate" >
       <Timeout duration="0" actionOnSI="blockHost" actionOnWorkload="retry"/>
       <Exit actionOnSI="blockHost" actionOnWorkload="retry"/>
       <Return controlCode="0" actionOnSI="keepAlive" actionOnWorkload="succeed"/>
       <Exception type="failure" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="retry"/>
      <Exception type="fatal" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="fail"/>
   </Method>
   <Method name="Invoke" >
      <Timeout duration="0" actionOnSI="restartService" actionOnWorkload="retry"/>
      <Exit actionOnSI=" restartService" actionOnWorkload="retry"/>
      <Return controlCode="0" actionOnSI="keepAlive" actionOnWorkload="succeed"/>
      <Exception type="failure" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="retry"/>
      <Exception type="fatal" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="fail"/>
   </Method>
   <Method name="SessionLeave" >
     <Timeout duration="0" actionOnSI="restartService"/>
     <Exit actionOnSI=" restartService"/>
     <Return controlCode="0" actionOnSI="keepAlive" customizedDebugAction="none"/>
     <Exception type="failure" controlCode="0" actionOnSI="keepAlive"/>
     <Exception type="fatal" controlCode="0" actionOnSI="keepAlive"/>
   </Method>
   <Method name="DestroyService" >
     <Timeout duration="15"/>
   </Method>
</Control>
```

Configuration example: Block the host when the service process exits during the invoke

```
<Control>
   ...
     <Method name="Invoke" >
        <Exit actionOnSI="blockHost"/>
     </Method>
</Control>
```

Configuration example: Do not rerun the task when a service times out

Configure a timeout value for the `onInvoke()` method and specify to fail the workload when the method times out.

```
<Control>
   ...
   <Method name="Invoke" >
       <Timeout duration="15" actionOnWorkload="fail" />
   </Method>
</Control>
```

Configuration example: Do not block hosts ever, under any situations

It is possible that in your environment you do not want hosts to be blocked under any circumstances. As a best practice, setting the system to not block hosts ever is not recommended. This is because if a service fails on a host, if the host is not blocked, the system may end up in an endless loop attempting to start the service on the same host on which it is always failing and continuously writing errors to the log files. In a very short time, your log files grow very large and take up too much disk space on your machine.

However, should you need to do this in your environment, you can accomplish this by setting all actionOnSI parameters to not block the host:

```
<SIM startUpTimeout="60" blockHostOnTimeout="false" blockHostOnVersionMismatch="false">
  …
</SIM>
<Control>
      <Method name="Register" >
         <Timeout duration="60" actionOnSI="restartService"/>
         <Exit actionOnSI="restartService"/>
    </Method>
    <Method name="CreateService" >
        <Timeout duration="0" actionOnSI="restartService"/>
        <Exit actionOnSI="restartService"/>
        <Return controlCode="0" actionOnSI="keepAlive"/>
        <Exception type="failure" controlCode="0" actionOnSI="restartService"/>
        <Exception type="fatal" controlCode="0" actionOnSI="restartService"/>
    </Method>
    <Method name="SessionEnter" >
        <Timeout duration="0" actionOnSI="restartService" actionOnWorkload="retry"/>
        <Exit actionOnSI="restartService" actionOnWorkload="retry"/>
        <Return controlCode="0" actionOnSI="keepAlive" actionOnWorkload="succeed"/>
        <Exception type="failure" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="retry"/
>
        <Exception type="fatal" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="fail"/>
    </Method>
    <Method name="SessionUpdate" >
        <Timeout duration="0" actionOnSI="restartService" actionOnWorkload="retry"/>
        <Exit actionOnSI="restartService" actionOnWorkload="retry"/>
        <Return controlCode="0" actionOnSI="keepAlive" actionOnWorkload="succeed"/>
        <Exception type="failure" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="retry"/
>
        <Exception type="fatal" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="fail"/>
    </Method>
    <Method name="Invoke" >
       <Timeout duration="0" actionOnSI="restartService" actionOnWorkload="retry"/>
       <Exit actionOnSI="restartService" actionOnWorkload="retry"/>
       <Return controlCode="0" actionOnSI="keepAlive" actionOnWorkload="succeed"/>
       <Exception type="failure" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="retry"/>
       <Exception type="fatal" controlCode="0" actionOnSI="keepAlive" actionOnWorkload="fail"/>
    </Method>
    <Method name="SessionLeave" >
       <Timeout duration="0" actionOnSI="restartService"/>
       <Exit actionOnSI=" restartService"/>
       <Return controlCode="0" actionOnSI="keepAlive"/>
       <Exception type="failure" controlCode="0" actionOnSI="keepAlive"/>
       <Exception type="fatal" controlCode="0" actionOnSI="keepAlive"/>
    </Method>
    <Method name="DestroyService" >
       <Timeout duration="15"/>
     </Method>
</Control>
```

Commands

Commands to display configuration

Command	Description
soamview app *app_name* -p	Once the application is registered, use `soamview app -p` to view the application profile xml. You can also view the xml configuration through the Platform Management Console.

Feature: Host blocking

Host blocking—a feature of application error handling—prevents Symphony from repeatedly trying to run a service on a host that does not have adequate hardware or software resources. You can configure host blocking to take effect on timeout or exit for each of your services, or when a service throws an exception or sends a specific return code.

About host blocking

When host blocking takes effect, Symphony creates a blocked host list for the application with which the service is associated. A host that appears on the blocked host list can no longer be used by the application until you intentionally unblock the host, or the application is re-registered or disabled and enabled again.

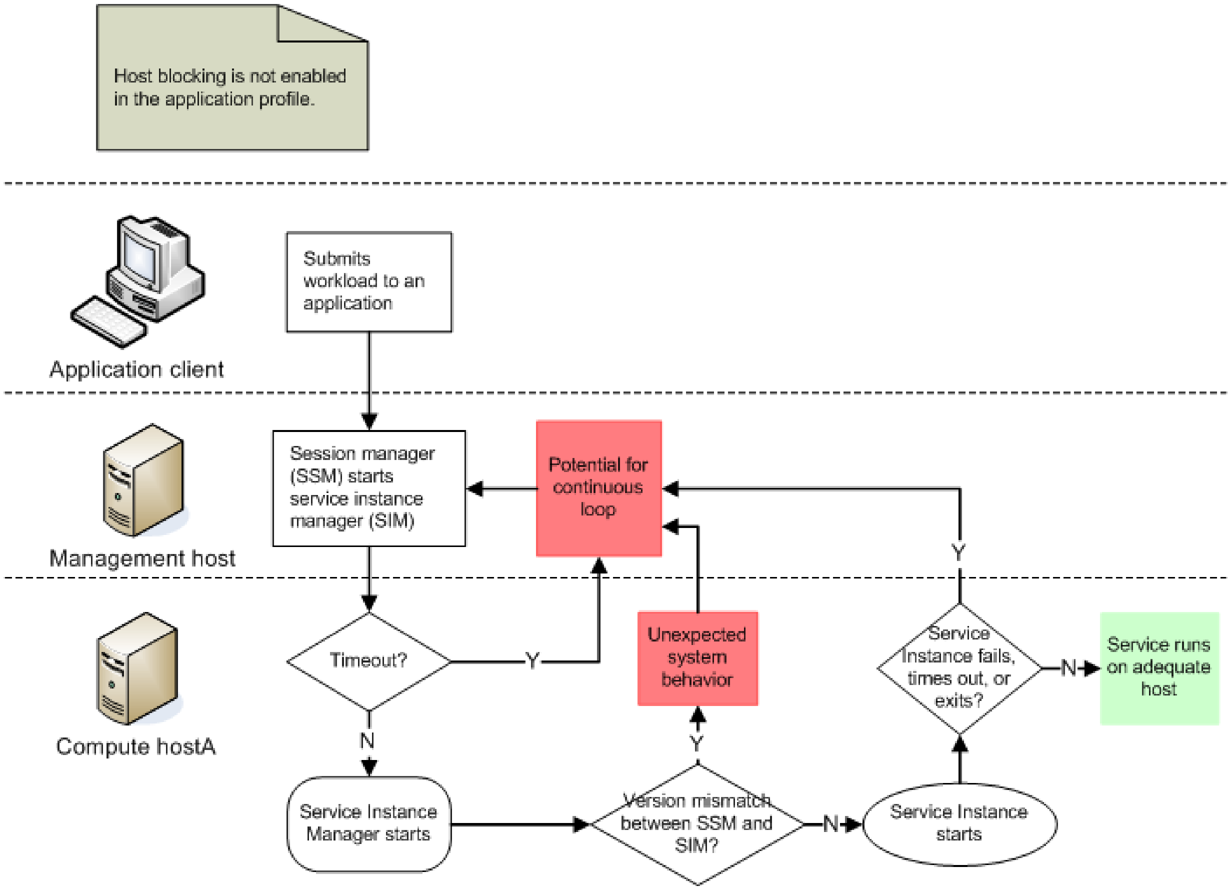
By default, host blocking is enabled for a version mismatch or communication timeout between the session manager and the service instance manager. You can also configure host blocking for a service instance error, a service instance exit, or a service instance method timeout. By default, host blocking is enabled for the following service instance methods:

Method	Event types
Register	• Timeout • Exit
CreateService	• Timeout • Exit • Failure exception • Fatal exception
SessionEnter	• Timeout • Exit

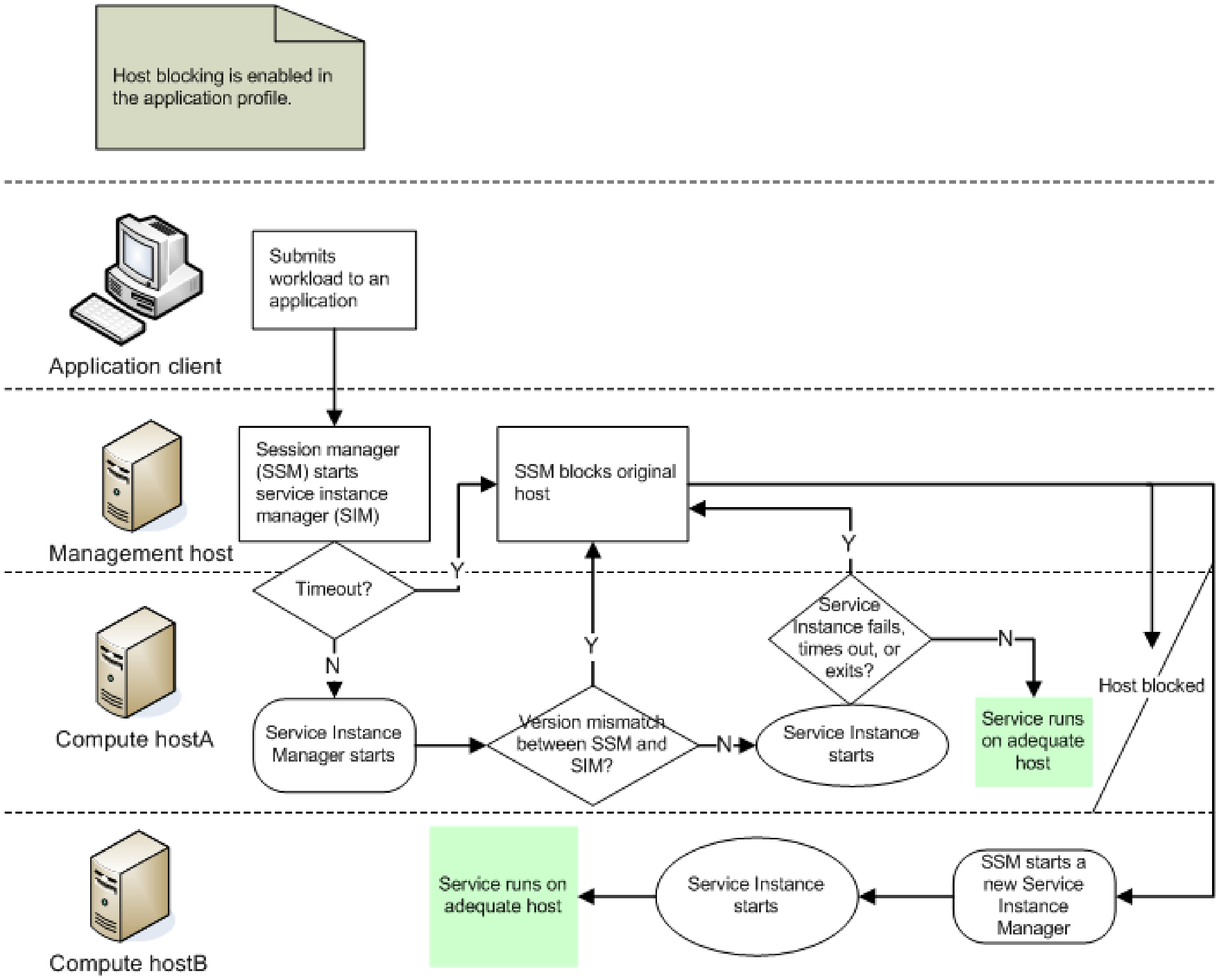
Method	Event types
SessionUpdate	- Timeout - Exit

The following illustrations show the benefits of using the host blocking feature.

Without host blocking (feature disabled)


Host blocking is not enabled in the application profile.
Application client
Submits workload to an application
Management host
Session manager (SSM) starts service instance manager (SIM)
Potential for continuous loop
Compute hostA
Timeout?
Y
N
Unexpected system behavior
Service Instance fails, times out, or exits?
Y
N
Service runs on adequate host
Service Instance Manager starts
Version mismatch between SSM and SIM?
Y
N
Service Instance starts

With host blocking enabled


Host blocking is enabled in the application profile.
Application client
Submits workload to an application
Management host
Session manager (SSM) starts service instance manager (SIM)
SSM blocks original host
Timeout?
Y
N
Compute hostA
Service Instance Manager starts
Version mismatch between SSM and SIM?
Y
N
Service Instance starts
Service Instance fails, times out, or exits?
Y
N
Service runs on adequate host
Host blocked
Compute hostB
Service runs on adequate host
Service Instance starts
SSM starts a new Service Instance Manager

Host blocking triggers

Host blocking triggers automatically when the session manager version on the management host does not match the service instance manager version on the comput host.

You can configure additional host blocking based on the requirements of your application so that Symphony triggers host blocking for any of the following reasons:

- A service method times out, exits or crashes, throws an exception, or returns certain control codes.
- The service instance manager does not communicate with the session manager before the configured timeout period expires (controlled by the startUpTimeout value).
- The service instance does not communicate with the service instance manager before the configured timeout period expires (controlled by the setting for the Register method actionOnSI attribute).

Slot blocking for Symphony DE

Symphony DE blocks slots—not hosts—under the same conditions that trigger host blocking for a production grid. Symptoms of blocked slots include fewer resources than expected or no resources serving your application, more tasks in the PENDING state, a slower rate of workload completion, and clients that hang. You can check for blocked slots by looking in the `ssm.`*`hostname.app_name`*`.log` file and searching for `WARN` or `ERROR` messages about blocked hosts. If you see a blocked host message, one or more slots might be blocked. You can unblock slots by disabling and then enabling the application or by restarting the DE cluster.

Scope

Applicability	Details
Operating system	• All host types that are supported by the Symphony system.
Limitations	• For Symphony DE, only slots are blocked.

Configuration to enable host blocking

Host blocking is enabled in the application profile for each application. You can configure host blocking at the service instance manager level, the service instance level, or both.

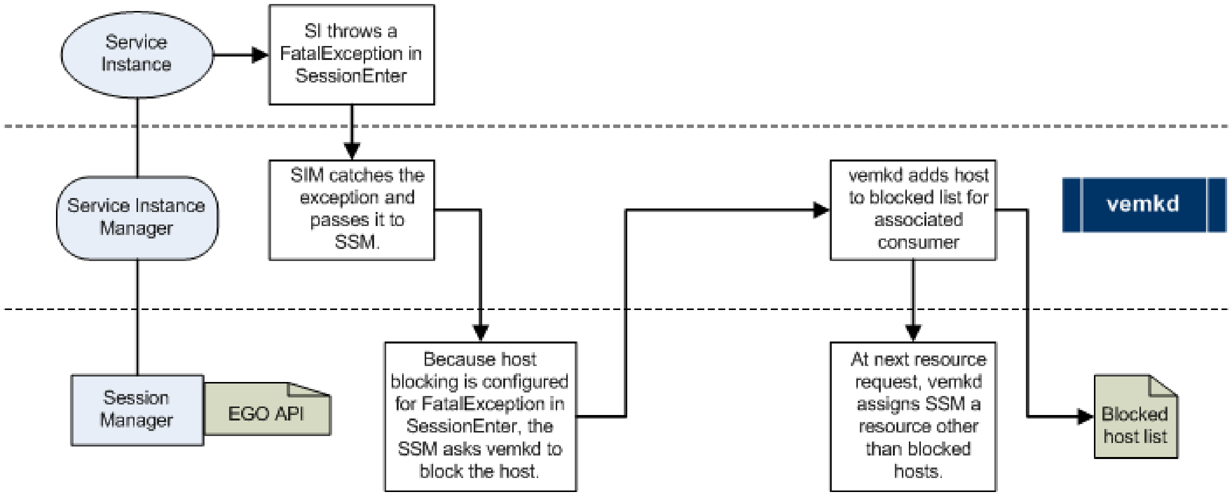
Section	Attribute name and syntax	Behavior
SOAM > SIM	blockHostOnTimeout="true"	• Enables host blocking for the application when the service instance manager times out while trying to communicate with the session manager. • Used with the startUpTimeout attribute.
	startUpTimeout="*seconds*"	• Number of seconds to wait for the service instance manager to communicate with the session manager. This attribute works in conjunction with blockHostOnTimeout. • When the process times out, the session manager requests a new host from EGO and tries to start a new service instance manager on the new host.
Service > Control > Method > Timeout	actionOnSI=blockHost	• When a timeout is reached on the method, terminates the running service instance on this host and does not use this host to start any other service instance for the application. • Used with the duration attribute. • You can specify the blockHost option for the following methods: • Register • CreateService • SessionEnter • SessionUpdate • Invoke • SessionLeave

Section	Attribute name and syntax	Behavior
Service > Control > Method > Exit	actionOnSI=blockHost	• When the service instance exits or crashes during execution of the method, the system does not use this host to start any other service instance for the application • You can specify the blockHost option for the following methods: • Register • CreateService • SessionEnter • SessionUpdate • Invoke • SessionLeave
Service > Control > Method > Return	actionOnSI=blockHost	• When the method returns normally or with a specified control code, terminates the running service instance on this host and does not use this host to start any other service instance for the application. • You can specify the blockHost option for the following methods: • CreateService • SessionEnter • SessionUpdate • Invoke • SessionLeave
Service > Control > Method > Exception	actionOnSI=blockHost	• When the specified exception (failure or fatal exception) occurs, terminates the running service instance on this host and does not use this host to start any other service instance for the application. • You can specify the blockHost option for the following methods: • CreateService • SessionEnter • SessionUpdate • Invoke • SessionLeave

Host blocking behavior

When host blocking is triggered, the system creates a blocked host list for the application. The following example illustrates the host blocking process triggered at the service instance level.

Example of the host blocking process


Service Instance
SI throws a FatalException in SessionEnter
Service Instance Manager
SIM catches the exception and passes it to SSM.
vemkd adds host to blocked list for associated consumer
vemkd
Session Manager
EGO API
Because host blocking is configured for FatalException in SessionEnter, the SSM asks vemkd to block the host.
At next resource request, vemkd assigns SSM a resource other than blocked hosts.
Blocked host list

Configuration to modify host blocking behavior

Not applicable. There are no attributes that change the way that host blocking works other than those attributes configured in the application profile.

Host blocking actions

Actions to monitor

You can monitor host blocking through the Platform Management Console (PMC), the command line, and through the Symphony log files located in the `logs` directory of `SOAM_HOME`. You can also trap SNMP events to receive notifications when a service triggers the system to block a host.

User	Action	Description
- Cluster administrator	From the Platform Management Console: **Symphony Workload > Monitor Workload > *application_name* > Blocked Hosts**	- Displays a list of blocked hosts for the selected application.
- Cluster administrator - Consumer administrator	From the command line: `egosh alloc view`	- Displays detailed information about all allocations, including the allocation ID, current users, consumer, resource groups, resource requirements, minimum and maximum slots requested, whether it has exclusive use of the host, names of the allocated hosts, and any blocked hosts.

You can find information about host blocking in the following log file:

Log file	Location	Event	Description

Session manager log file: `ssm.`*`host_name`*`.`*`app_name`*`.log`	Linux/UNIX:
	`$SOAM_HOME/logs`
	Windows:
	`%SOAM_HOME%\logs`
		SOA_SERVICE_BLOCKED	Error level message that indicates that host blocking has occurred.

Actions to control

Typically, a cluster administrator removes a blocked host when the host has been modified—by means of a software or hardware upgrade, for example—to meet the requirements of the service. A host can be removed from the blocked host lists in one of two ways:

- Directly from the Platform Management Console (PMC)
- Indirectly, by disabling and re-enabling the application associated with the blocked host

User	Action	Behavior
• Cluster administrator	From the Platform Management Console: **Symphony Workload > Monitor Workload > *application_name* > Blocked Hosts > *host_name* > Actions > Remove from Blocked Hosts**	• The system removes the host from the blocked host list • The application can start a service on the previously blocked host
• Cluster administrator	From the Platform Management Console: **Symphony Workload > Configure Applications > *consumer_name* > *application_name* > Actions > Disable**	• Disables the application, which clears the blocked host list for the disabled application • No clients can be served by the disabled application
• Cluster administrator • Consumer administrator • Consumer user	From the command line: `soamcontrol app disable` *application_name*	• Disables the application, which clears the blocked host list for the disabled application • No clients can be served by the disabled application • For information about how to use the `soamcontrol` command to disable and enable applications, see the *Reference*
• Cluster administrator • Consumer administrator	From the Platform Management Console: • **Symphony Workload > Configure Applications > *application_name* > Basic Configuration > Save** • **Symphony Workload > Configure Applications > *application_name* > Advanced Configuration > Save**	• The system first disables and then re-registers the application, which clears the blocked host list for the modified application

For Symphony DE, you can unblock slots by disabling and then enabling the application, or by restarting the DE cluster.

User	Action	Behavior
- Developer	From the command line: `soamcontrol app disable` *application_name*	- Disables the application, which unblocks slots for the disabled application - No clients can be served by the disabled application
	`soamcontrol app enable` *application_name*	- Enables the application, which can start services on any previously blocked slot
- Developer	Windows: - Right-click on the Symphony DE tray icon and choose **Stop Symphony DE on all hosts**. Once the DE cluster shuts down, right-click on the Symphony DE tray icon and choose **Start Symphony DE on all hosts**. Linux/UNIX: - `soamshutdown` - `soamstartup`	- Shuts down and then restarts Symphony DE - Unblocks slots for all applications running on the DE cluster

Actions to display configuration

User	Command	Behavior
- Cluster administrator - Consumer administrator	From the Platform Management Console: - **Symphony Workload > Configure Applications >** ***application_name*** **> Basic Configuration** - **Symphony Workload > Configure Applications >** ***application_name*** **> Advanced Configuration**	- Displays application profile settings for the selected application
- Cluster administrator - Consumer administrator - Consumer user	From the command line: - `soamview app` *app_name* `-p`	- Displays application profile settings for the selected application

You can also view an application profile using an XML editor.

Feature: Service interrupt handler

This feature enables developers to respond appropriately to any events that interrupt the running of a service instance during execution of a task.

Scope

Applicability	Details
Operating system	• Windows • Linux • Solaris
Limitations	None

About service interrupt handling

A *service interrupt* is as any event that is propagated from the middleware to the service instance indicating an interruption has happened in a running task. Interruptions can arise when:

- A session is killed
- A session is suspended
- A resource is reclaimed
- A task is killed
- An application is unregistered or disabled
- A middleware component is not available

This feature allows interrupt handling to move from a polling model to an event-driven model. The polling model is still available for backward compatibility.

When a service implements the Service Interrupt handler, the service is informed as soon as the interrupt occurs. Service developers no longer need to have a separate monitoring thread for interrupts to determine when an interrupt has occurred.

Only task-level events are propagated to a running service instance. An interrupt is not delivered to a running service instance if no task is running (`onInvoke` is not called).

Once the `onServiceInterrupt()` handler is triggered, it is passed the current service context object. To get more details about the actual interrupt (for example, to retrieve the event and grace period in the interrupt handler), the service can call the getLastInterruptEvent method on the service context object.

Supported interrupts

The Service Interrupt handler can send the following interrupts to a service instance at any time:

Task killed interrupt	This interrupt is sent as a result of an administrative operation to kill a task or a session. When called, the `onInvoke()` handler is expected to return from its processing by the time the grace period expires. If the grace period expires with no reply from the service, the service is terminated. • C++: InterruptTaskKilled • C# .NET: InterruptEventCode.TaskKilled

	• Java: InterruptEvent.TASK_KILLED
Task suspended interrupt	This interrupt is sent as a result of an administrative operation to suspend a session or when a resource running the service instance is being reclaimed. When called, the `onInvoke()` handler is expected to return from its processing by the time the grace period expires. If the grace period expires with no reply from the service, the service is terminated. If the `onInvoke()` handler successfully finishes before the grace period expires, the task is considered done. • C++: InterruptTaskSuspended • C# .NET: InterruptEventCode.TaskSuspended • Java: InterruptEvent.TASK_SUSPENDED

Error handling

Interrupts do not have any error handling associated with them. Any exception thrown while handling this method is ignored by the middleware. The middleware reports the exception to the service instance so that the service instance can take the necessary actions to handle the exception.

Service interrupt handling API

The following handlers are introduced to the service:

C++

```
virtual void onServiceInterrupt (ServiceContextPtr&
serviceContext)
```

C# (.NET)

```
public override void OnServiceInterrupt(ServiceContext
serviceContext)
```

Java

```
public void onServiceInterrupt (ServiceContext
serviceContext) throws SoamException
```

CHAPTER

5

Using Eclipse as Your Development Environment

Feature: Symphony plug-in for Eclipse

The Symphony plug-in for Eclipse facilitates the software development cycle for Symphony applications. It provides Java code generation capabilities as well as service and application management tools for the Eclipse IDE.

Scope

Operating system	Follow the **System requirements** link at the Platform Knowledge Center for a list of supported operating systems
JDK version	• 1.5
Eclipse version	• Tested on 3.2.2 and 3.3.0
Limitations	• Some dialogs may not display correctly on Linux Platforms due to Eclipse-GTK integration issues. • Some windows will not show the minimize button enabled on some Linux Platforms due to known problems in GTK implementations. • Some dialogs may display with a redundant input edit control on Linux Platforms due to Eclipse-GTK integration issues. • The 32-bit version of Eclipse is not supported on a 64-bit Linux host. A 64-bit version of Eclipse is available for download from the Eclipse site.

About the Symphony plug-in for Eclipse

The plug-in eases the task of application coding where client and service code must be written or adapted for the Symphony API. The plug-in automates some of the coding effort, thereby reducing or eliminating errors. Once the Java project wizard has created a framework of generated code, all you need to do is add the application logic.

The automation introduced by the plug-in extends to the creation and maintenance of Symphony applications. For example, the tasks of deploying a service or updating an application profile is simplified through the Symphony DE Platform Management Console (PMC), which is tightly integrated to the Eclipse environment. Depending on what type of Symphony operation you want to perform, the appropriate screen is presented to guide you through the operation.

The plug-in also features extensive online documentation to support your Symphony programming activities such as informative comments in the generated code, the Symphony Java API reference documentation, and a tutorial.

Installing the Symphony plug-in

This section describes the steps for installing the Symphony plug-in into the Eclipse IDE.

On Windows

1. Install the Symphony DE package.
2. Launch Eclipse.
3. From the Eclipse menu, select Help > Software Updates > Find and Install.

   The Install/Update dialog displays.
4. Select Search for new features to install. Click Next.

5. Click New Local Site.
6. Browse to `%SOAM_HOME%\4.1\binary_type\plugins`.
7. Select eclipse.
8. Click OK.

   The Edit Local Site dialog displays.
9. Click OK.

   plugins/eclipse is added to the list of update sites.
10. Click Finish.

    The Updates dialog displays.
11. Select plugins/eclipse. Click Next.

    The Install dialog displays.
12. Select I accept the terms in the license agreement. Click Next.
13. Click Finish.

    The update manager installs the plug-in and the Install/Update message displays.
14. Click Yes to restart Eclipse. For an introduction to Symphony code development with Eclipse, refer to the *Developing a Symphony application with Eclipse* tutorial.

On Linux

1. Install the Symphony DE package.
2. Launch Eclipse.
3. From the Eclipse menu, select Help > Software Updates > Find and Install.

   The Install/Update dialog displays.
4. Select Search for new features to install. Click Next.
5. Click New Local Site.
6. Browse to `$SOAM_HOME/4.1/binary_type/plugins`.

   For example:

   `$SOAM_HOME/4.1/linux2.6-glibc2.3-x86/plugins`
7. Select eclipse.
8. Click OK.

   The Edit Local Site dialog displays.
9. Click OK.

   plugins/eclipse is added to the list of update sites.
10. Click Finish.

    The Updates dialog displays.
11. Select plugins/eclipse. Click Next.

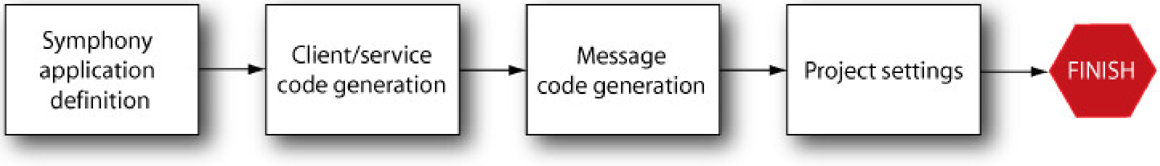
    The Install dialog displays.
12. Select I accept the terms in the license agreement. Click Next.
13. Click Finish.

The update manager installs the plug-in and the Install/Update message displays.

14. Click Yes to restart Eclipse. For an introduction to Symphony code development with Eclipse, refer to the *Developing a Symphony application with Eclipse* tutorial.

Project wizard

The New Project wizard generates Java client, I/O message, and service code based on the information provided through the GUI screens. The result is a framework of generated code that can either be incorporated into your code or used as a basis for your application. Once your service code is complete, you can deploy it to the DE environment using the plug-in's Symphony Service Packing Utility.


Symphony application definition
Client/service code generation
Message code generation
Project settings
FINISH

To launch the wizard, select File > New > Project > Symphony from the Eclipse menu. You can create either a blank project or a project with generated code.

The following paragraphs describe the major steps in the flow of the New Project wizard (with generated code).

Application definition

In this initial step, you provide the name of your application and Java package name for the generated classes. If you do not enter a package name, the wizard generates the code in the default namespace.

If you decide to change the application name after you have generated the code with the project wizard, the CodeGen icon on the Eclipse toolbar provides a dialog where you can change it.

Important:

If you change the application name using this method, you must manually update the application name in the client code.

Client/service code generation

In this step, you assign names to your client and service classes, and define whether your client will receive messages synchronously or asynchronously.

Message code generation

This is where you define the messages that will be used within your application, e.g., input and output message classes. If you already have a class that is serializable then you can include it as an element of this message. If you include a serializable class that doesn't exist, you will need to add the class to the project before compiling it.

Project configuration

By default, the wizard adds the JavaSoamApi.jar and JRE system library to the dependency list for Symphony Java applications. Additional Java project configuration is possible through the native Java configuration dialog in Eclipse.

Symphony operations

In addition to client and service code generation, the plug-in supports a number of operations integral to the creation and maintenance of Symphony applications. Most of these operations are performed via the Symphony DE PMC, which is automatically launched when the applicable operation is selected. To access Symphony operations from Eclipse, select Symphony > Symphony Operations from the menu.

Creating a service package

The Symphony Service Packaging Utility facilitates service package creation and deployment into the Symphony DE environment. Once you have entered the service class name, package name, and package path, and included the necessary file(s), you can invoke the utility to create and validate the service package. If you already have an application profile for this service, the utility can also deploy the service for you.

Adding an application

The Symphony DE PMC enables you to add a Symphony application to the DE environment by using either an existing application profile or creating a new application profile with basic settings. Once your application is set up, it is automatically registered with Symphony.

Editing an application profile

The Symphony DE PMC features an application profile editor for most basic and advanced configuration. Once updated, the application profile can overwrite an existing profile or be exported to a file.

Re-registering with an external application profile

For most basic application settings, you can edit the application profile directly with the Symphony DE PMC. However, there are some settings that cannot be edited by the GUI. In these cases, you must export the application profile and use an XML or text editor to edit the profile, then import it back to the application, i.e., re-register the application.

Monitoring workload

This feature allows you to view session and task status, application properties, and application profile via the Symphony DE PMC.

Configuring service debug settings

The service replay debugger allows you to replay actual events that occurred in your service instance when you ran your application in Symphony DE. This feature provides a debugger to step through the service code and find any errors in your service logic or in the environment.

Selecting this operation from the Eclipse menu launches the application profile editor in the Symphony DE PMC. Select Advanced Configuration from the dropdown list. You can then configure debug settings in the Error Handling section of the GUI. Refer to the Service Replay Debugger feature reference for further details.

Configuring the Symphony DE Platform Management Console for the Eclipse IDE

When the plug-in is installed, it connects to the Symphony DE PMC through the default port specified in the vem_resources.conf file. Should the port be reconfigured in the file, you must specify the new port via the Platform Management Console icon on the Eclipse toolbar.

CHAPTER

6

Developing Admin Clients

Tutorial: Admin Web Service client tutorial

Goal

This tutorial walks you through an example of Java client application code that changes the priority of specified sessions via the Admin Web Service interface. This tutorial was prepared for users that are already familiar with Web Services.

For information about Web Service concepts, refer to the Web Service clients section of the Application Development Guide in the Knowledge Center.

At a glance

1. About client - server interactions
2. Review and understand the example

Prerequisites

- JDK 1.4 or higher (for Java development)
- Third-party WSDL - client stub generating tool that complies with the following standards:
    - WSDL 1.1
    - SOAP 1.1
    - Web Service Security UsernameTokenProfile 1.0 (defined in WS-Security 1.0 specification (UsernameToken part only); refer to http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd.)

  The following tools have been tested for Java and .NET:

    - Java: Axis2 0.93
    - .NET: Microsoft WSE version 3.0

About client - server interactions

To better understand how the client connects to the Symphony, let’s look at the sequence of events from start-up.

1. When EGO starts up, it launches the service controller.
2. The service controller starts the session director server as a service.
3. Upon startup, the session director listens for incoming connections from clients. The session director also registers with EGO as a client and uses its connection URL as the client description.
4. The Symphony client opens a connection to EGO.
5. The Symphony client retrieves the connection URL to session director by passing the session director client name in an API call to EGO and retrieving the client description (connection URL).
6. The Symphony client connects to the session director.

Steps for developing Symphony Web Service clients

Here is a suggested high-level methodology for developing Web Service clients.

General development sequence

1. Read the WSDL documentation to understand the API.

2. Use the appropriate tool (that supports the required specifications) to generate language-specific binding.
3. Use the generated code to write the client application.

Symphony-specific development sequence

1. Find where the Session Director is running.
2. Log on to the cluster and obtain a credential.
3. Use the Session Director's Admin API calls.

Review and understand the example

We will review an example of client code to show how you can create a Java client application that modifies the session priority for a given application. The Java client code in this example is representative of the code you would use to access the proxy stub generated from Axis2.

Locate the documentation

You can access additional documentation such as the *Web Services WSDL Reference*, *Java API Reference*, the *Platform Symphony Reference*, and the *Error Reference* from the Platform Symphony or Symphony DE Knowledge Centers.

Windows

- From the Start menu, select Programs > Platform Computing > Symphony Developer Edition 4.1.0 > Developer Knowledge Center

Linux

- `$SOAM_HOME/docs/symphonyde/4.1/index.html`

What the example does

Using this tutorial, you do the following:

- List all the applications registered with Symphony
- List all the sessions with the specified application
- Change the priority of all the sessions with the specified tag
- List all the open sessions again to demonstrate that the priority has changed
- Handle session director failover.

Step 1: Initialize the client

Create an EGO client object by calling the EGOclient constructor and passing the EGO Web Service Gateway URL as an input argument. Use the initializePorts() method to set up a connection to each web service interface (port) that is required for this tutorial.

The session director (one per cluster) is responsible for authenticating clients and processing their requests. For the client to connect to the session director, it must know its URL. To get the session director's URL, pass the client name "SD_ADMIN" to the locate() method of the egoClient object. This method connects to the EGO web service gateway and retrieves all the client information associated with the session director including its URL, which is stored in the client description property of the ClientInfo object. The session director URL is in the form of domain name and port number. For more information about the locate() method, refer to the Web Services WSDL Reference in the Knowledge Center.

Prepend the communication protocol (http://) to the URL to complete it. Create a new proxy object and initialize it with the session director URL. This sets up a connection to the session director via a web service interface (port).

```
...
public void initialize(String egoUrl) throws Exception
{
    egoClient = new EGOclient(egoUrl);
    egoClient.initializePorts();
    String sdLocate = locateSD();
    if(0 == sdLocate.length())
    {
        throw new Exception("Cannot get the SD location successfully. Check if the URL
        of Web Service Gateway <" + egoUrl + "> is valid and the state of Web Service
        Gateway is <STARTED>.");
    }
    String sdUrl="http://" + sdLocate;

    System.out.println(sdUrl);
    soamStub = new SoamPortTypeStub(null, sdUrl);
}
...
```

```
...
public String locateSD()
{    String SdUrl="";    ClientInfo[] cinfos = egoClient.locate("SD_ADMIN");    if(null !=
cinfos)    {        if (cinfos.length > 0)
        {        SdUrl = cinfos[0].getClientDescription();        }    }    return SdUrl;}
...
```

Step 2: List all the applications registered with Symphony

To view a list of all registered applications, we pass an empty string to the viewApp() method.

For the client to interact with Symphony, it must first be authenticated by the session director. This requires that the client present a security credential. To acquire the credential, set up the security header using the setSecurityHeader() method. The logon method uses a plain text username and password that are sent over an SSL-enabled connection. The method returns the encrypted credential, which is stored in security document securityDoc. For more information about the logon() method, refer to the Web Services WSDL Reference in the Knowledge Center.

The following XML snippet demonstrates how the actual security header with username and password would appear on the wire.

```
<oas:Security xmlns:oas="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
secext
-1.0.xsd">
<wsse:wsse:UsernameToken
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secex
t-1.0.xsd">
  <wsse:Username>Admin</wsse:Username><wsse:Password
 wsse:Type="wsse:PasswordText">Admin</wsse:Password>
  </wsse:wsse:UsernameToken>
</oas:Security>
...
```

Since the Symphony admin Web Service uses document-style binding, the SdViewAppDocument and SdViewAppResponseDocument classes represent the XML documents that are exchanged between the Web Service and the client. The viewAppReq and resp classes represent the data.

An object of type SdViewAppDocument (viewAppReqDoc) and securityDoc, which has been initialized with the encrypted credential, is passed to the local proxy method, sdViewApp(). The Web Service returns the list of registered applications and the result is printed out to the console using the overloaded print() method.

```
public void viewApp(String appName) throws RemoteException
{
    setSecurityHeader();
    SdViewAppDocument viewAppReqDoc = SdViewAppDocument.Factory
    .newInstance();
    SdViewAppDocument.SdViewApp viewAppReq = viewAppReqDoc
    .addNewSdViewApp();
    viewAppReq.setAppName(appName);
    SdViewAppResponseDocument respDoc = soamStub.sdViewApp(viewAppReqDoc,
    securityDoc);
    SdViewAppResponseDocument.SdViewAppResponse resp = respDoc
    .getSdViewAppResponse();
    AppAttribute[] appVector = resp.getAppAttrVector().getItemArray();
    for (int i = 0; i < appVector.length; i++)
    {
        print(appVector[i]);
    }
}
...
```

```
 public void setSecurityHeader()
    {
        String creds = egoClient.logon("Admin", "Admin", null);
        String credential = egoClient.logon(null, null, creds);
        securityDoc = SecurityDocument.Factory.newInstance();
        Security securityReq = securityDoc.addNewSecurity();
        securityReq.setCredential(credential);
    }
...
```

Step 3: List all the sessions associated with a specific application

To view a list of all sessions associated with a specific application, we pass the application name and a string ("0") for session ID to the viewSession() method.

Set up the security document securityDoc by calling the setSecurityHeader() method; see Step 2: List all the applications registered with Symphony.

Create the request document viewSessionReqDoc and its associated request message object viewSessionReq. Initialize the viewSessionReq object with the application name and session ID that, in this case, is set to "0" to denote all sessions.

Call the local proxy method sdViewSession() and pass the request document viewSessionReqDoc and securityDoc as input arguments. The Web Service returns a list of all open sessions including session attributes, which is printed out to the console using the overloaded print() method.

```
...
public void viewSession(String appName, String sessionId) throws RemoteException
{
    setSecurityHeader();
    SdViewSessionDocument viewSessionReqDoc = SdViewSessionDocument.Factory
    .newInstance();
    SdViewSessionDocument.SdViewSession viewSessionReq = viewSessionReqDoc
    .addNewSdViewSession();
    viewSessionReq.setAppName(appName);
    viewSessionReq.setSessionId(Long.parseLong(sessionId));
    String filter="state=\"open\"";
    viewSessionReq.setFilter(filter);
    SdViewSessionResponseDocument respDoc = soamStub.sdViewSession(
    viewSessionReqDoc, securityDoc);
    SdViewSessionResponseDocument.SdViewSessionResponse resp = respDoc
    .getSdViewSessionResponse();
    SessionAttribute[] sessionVector = resp.getSessionAttrVector()
    .getItemArray();
    if (sessionVector.length > 0)
    {
        for (int i = 0; i < sessionVector.length; i++)
        {
            print(sessionVector[i]);
        }
    }
    else
    {
        System.out.println("No sessions found.");
    }
}
...
```

Step 4: Change the priority of all the sessions with the specified tag

The modSession() method accepts four input arguments: application name, session ID, session tag, priority. All these input arguments, with the exception of session ID, are provided as arguments when you run the main() program. The session ID is initialized to 0 to indicate to the middleware that you want to modify the priority for sessions that share the specified session tag.

Set up the security document securityDoc by calling the setSecurityHeader() method; see Step 2: List all the applications registered with Symphony.

Create the request document modSessionReqDoc and its associated request message object modSessionReq. Initialize the modSessionReq object with the application name, session ID, session tag, and the new priority.

Call the local proxy method sdModSession() and pass the request document modSessionReqDoc and securityDoc as input arguments. The Web Service modifies the priority

```
...
public void modSession(String appName, String sessionId, String filter,
String attrs) throws RemoteException
{
    setSecurityHeader();
    SdModSessionDocument modSessionReqDoc = SdModSessionDocument.Factory
    .newInstance();
    SdModSessionDocument.SdModSession modSessionReq = modSessionReqDoc
    .addNewSdModSession();
    modSessionReq.setAppName(appName);
    modSessionReq.setSessionId(Long.parseLong(sessionId));
    modSessionReq.setFilter(filter);
    modSessionReq.setModStr(attrs);
    SdModSessionResponseDocument respDoc = soamStub.sdModSession(
    modSessionReqDoc, securityDoc);
    SdModSessionResponseDocument.SdModSessionResponse resp = respDoc
    .getSdModSessionResponse();
}
...
```

Step 5: Handle session director failover

If the system is configured for the session director to fail over on different hosts, the Web Service client has to be written in a way to ensure that the request is sent to the correct session director URL. In the following example, code has been added to the example from step 3 to handle session director failover situations.

First, we initialize the retry variable so that a reconnection to the session director is attempted up to three times. The sdViewSession() method is called and if it succeeds, the variable is reset to zero and the loop exits. If the method call fails, relocateSD() makes an attempt to find the new host that the session director is running on. If the URL is found, it is passed to the soamStub object and the sdViewSession() method call is attempted again; otherwise, a RemoteException is thrown indicating that the session director cannot be found.

```
    ...
    setSecurityHeader();
    SdViewSessionDocument viewSessionReqDoc =
    SdViewSessionDocument.Factory.newInstance();
    SdViewSessionDocument.SdViewSession viewSessionReq =
    viewSessionReqDoc.addNewSdViewSession();
    viewSessionReq.setAppName(appName);
    viewSessionReq.setSessionId(str2sessionId(sessionId));
    int retry = 3; //retry 3 times
    while(retry > 0)
    {
        try
        {
            SdViewSessionResponseDocument respDoc =
            soamStub.sdViewSession(viewSessionReqDoc,securityDoc);
            retry = 0;
        }
        catch(java.rmi.RemoteException re)
        {
            reLocateSD();
            if (soamStub == null)
            {
                throw re;
            }
            else
            {
                retry--;
            }
        }
    }
    SdViewSessionResponseDocument.SdViewSessionResponse resp =
    respDoc.getSdViewSessionResponse();
    SessionAttribute[] sessionVector = resp.getSessionAttrVector().getItemArray();
...
public void reLocateSD()
{
    String sdUrl="http://" + locateSD();
    soamStub = new SoamPortTypeStub(null, sdUrl);
}
    ...
```

CHAPTER

7

Running Executables

Feature: Execution tasks integration

This feature supports the running of remote execution tasks using the Symphony infrastructure. An execution task is a child process executed by a Symphony service instance using a command line specified by a Symphony client.

Scope

Operating system	Linux, Windows, and Solaris hosts supported by Symphony
Limitations	If a symexec fetch command is in progress while a fetch or send command is issued for the same session from another command prompt window, the original fetch operation will abort and the session will detach from the original client.

About the Symphony execution task feature

The Symphony execution task feature provides the ability to deploy existing executables in a grid environment, thereby realizing the benefits of Symphony without necessarily redeveloping your application code. If you have existing executables but are developing new clients, the Symphony SDK API can be used to start and control the remote execution of executable files. Each application consists of an execution service and an executable file that are distributed among compute hosts.

The execution service is common to all execution applications and is primarily responsible for starting the remote execution tasks and returning results to the client. When you install Symphony or Symphony DE, the execution application is pre-deployed but disabled.

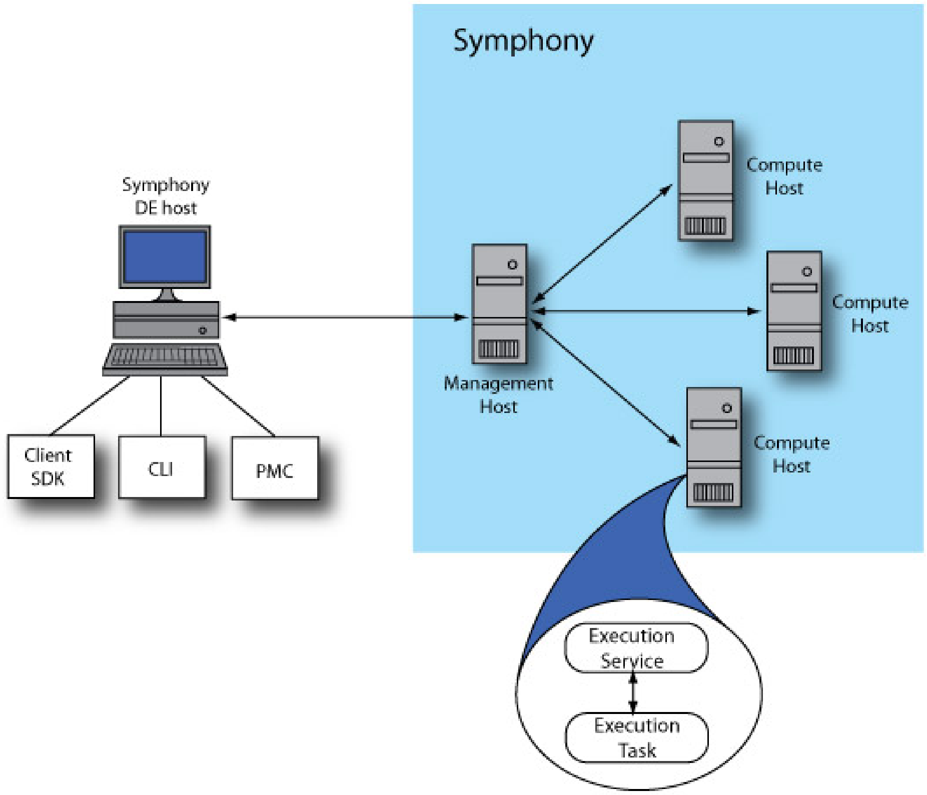
Here are the characteristics of a Symphony execution application:

- only exit codes from the execution tasks are returned to the client. Note that the execution service, which manages the execution tasks, can still return exceptions
- the execution task is started and stopped by the service process for every task, which has more overhead than the method invocation of the ordinary Symphony task within the service process.

There are three ways to run, control, and monitor execution tasks from the client host:

- Platform management console (PMC)
- command line interface (CLI)
- client SDK

Each interface offers a different level of functionality. For example, with the client SDK you can spawn a second thread to issue a non-blocking fetchTaskStatus() call while the client performs other functions. Or you can use the CLI for scripting execution task commands. For more information, refer to the Interfaces section of this chapter.



For the execution application to work, the executable files must reside on the same compute host as the execution service or at least be accessible from the compute host through a file sharing system or remote shell. Symphony's service deployment tools can be used to package the executable files and dependencies, and distribute them to the compute hosts.

To help you decide whether the execution application is right for your needs, here is the criteria you should use:

- you have an existing executable and want to reuse it without changing it to integrate with the Symphony API
- you want to write a script to run the executable from on Symphony
- you have no access to the source code and cannot change it

Application execution flow

To help grasp the concepts, let's look at the functional flow of an execution application.

1. The client application creates a Symphony execution session and passes the following optional session-level data to the execution service:

    - array of environment variable strings in "name=value" pairs format (can be empty)
    - pre-execution command string (can be empty)
    - post-execution command string (can be empty)

    The execution service ensures that if the pre-execution command completes successfully, the post-execution command will always be executed, even if errors occur in the user's command during the session.

2. The client application sends the execution tasks to the execution service., which includes:
    - an executable command string with arguments
    - optional execution task context data including environment variable strings in "name=value" pairs format and pre-/post- execution commands
3. Upon receiving the input message on the service side, the execution service spawns a new process based on the execution task data.

    While the execution task process is running, the execution service periodically checks for interruptions and suspends or aborts the running process if it detects an administrative action such as session suspend or abort was issued by the CLI or PMC.
4. When the execution task has completed its process, the exit code of the process is sent back to the client in the execution task status. The error handling associated with this exit code is configurable in the application profile.

Note:

The exit code of pre- and post-commands is only logged, and not sent back to the client.

Execution service

The execution service implements the process execution logic as a Symphony service and interacts directly with the service instance manager (SIM).

To allow the execution task to use some unique identities, the service side execution environment is supplemented with the following additional environment variables:

- SYM_SERVICE_NAME = execution service name
- SYM_SERVICE_PID = execution service instance OS process identifier
- SYM_SESSION_ID = session identification number
- SYM_TASK_ID = task identification number

The execution service logs session and command start-up and shutdown audit information in a log file. The path for log files is configurable in the application profile. By default, the log is set to the INFO level, which includes the following data:

- date/time stamp
- local time zone
- host name
- service instance process identifier
- session identifier (for session pre- and post- commands)
- execution task identifier (for command-level executions)
- execution level (session or task)
- execution context (pre-, post-, or command execution)

Interfaces

This section describes the three interfaces that are available for sending execution tasks.

Client SDK

The Symphony execution task feature extends the existing Symphony API classes for C++ and Java languages:

Classes:

- ExecutionSession: a class to enable the client to manage its execution workload
- ExecutionEnumItems: container class for ExecutionStatus objects
- ExecutionStatus: container class for the status of an execution task result
- ExecutionSessionContext: container class for environment variables, session type, and pre-/post- commands for session-level context
- ExecutionCommandContext: container class for environment variables and pre-/post- commands for task-level context

Command line interface

Use symexec for the execution application operations. The symexec utility supports the following commands:

- create - creates an execution session in which to run an execution task. The session stays open until it is explicitly closed (detachable session)
- send - sends an execution task command in the execution session
- fetch - fetches the statuses of finished execution tasks in the execution session
- close - closes the execution session
- run - creates an un-detachable execution session, runs an execution task, waits and then retrieves the result, and closes the execution session.

For more details such as command syntax and options, refer to the Command Reference.

Platform management console

The management console implements a GUI panel that allows a single execution task to be sent per session. The interface also supports the entry of associated pre- and post- commands, as well as environment variables and their values. Basically, the functionality offered by this panel is identical to the `run` subcommand of the symexec CLI where "Guest" is the username and password. If you need to send multiple concurrent execution tasks, additional GUI dialogs must be opened.

The management console can retrieve log files that are created by the execution service for each execution session.

For more details, refer to the online help provided with the management console.

Configuration

Application profile

Every application requires an application profile to dynamically configure the application's behavior within Symphony. An application profile for the default symexec application is already registered (but disabled) when Symphony is installed. There is also a second application profile, specific to execution applications, included in the Symexec sample that is packaged with Symphony.

The execution service is common to all execution applications. In cases where different consumers are running execution applications, the name of the application specified in the application profile must be unique. Also, each application profile name must be unique in the cluster and associated with only one consumer. At any time, there can be only one enabled application per consumer.

For the execution application to work properly, the following key parameters must be configured in the application profile:

- recoverable

  Set this flag to true since the execution application must be recoverable to support suspend/resume operations.
- suspendGracePeriod and taskCleanupPeriod

  Set both parameters so that the value is greater than the time it takes to complete the post-execution command that is executed by the service's `onSessionLeave()` method. This time period allows for post-execution cleanup.
- controlCode

  In the application profile, the SessionEnter and SessionLeave Method elements correspond to the service methods `onSessionEnter()` and `onSessionLeave()`. It is within these methods that the session-level pre- and post- execution commands are executed. Similarly, the Invoke Method element corresponds to the `onInvoke()` service method where task-level pre- and post- execution commands, and the execution task's command itself are executed. For the default execution application, its application profile is configured so that if the command's exit code has a value of zero, it will be treated as successful completion and any other value will be treated as a failure of the corresponding execution task.

  When an execution task's command or session-level pre- and post- command is able to start and finish, the execution service stores the command's exit code as a Symphony service context's control code and then adds 1 to the control code. Therefore you may configure the control code for the Return event in the application profile with a value of 1 greater than the exit code you are expecting from the given command. For example, the default execution application is configured that when the command executes successfully with an exit code of 0, it sets the control code for this result to 1 (exit code +1) so that Symphony processes the result as a successful action.

  Note:

  The exit codes for task-level pre- and post- execution commands are not stored so consequently the handling of these exit codes cannot be configured.

  The default value for all unspecified control codes is 0. It means that if you have specified control code equal to 1 to indicate successful action, all failures or system errors that do not have a corresponding control code in the application profile will be automatically associated with control code 0.

  If the command cannot be started by the operating system, the execution service sets the control code to the system error number returned by the operating system. You can configure the Failure events in the application profile with control codes to handle specific exceptions. In this case, if the matching control code is configured as a failure, the exception message with the error code will be propagated back to the client. If an exception occurs with a control code that is not configured, it will automatically be associated with the action for control code 0.

  If your execution application has more than one possible exit code to indicate successful completion, you must configure these multiple control codes in the application profile associated with the successful action.

If you are using multiple application profiles for multiple execution applications, you can configure the logDirectory parameter in each profile so that each application stores logs in a unique location. Another way of organizing log files per application is to use different service names and update the respective application profile with the new service name. All application profiles use the service name in the sub directory path (subDirectoryPattern) for logging and this path is fixed on the service side for compute hosts. Changing the service name for each application profile is another way to ensure a unique location for each application’s log files.

Logging

Logging that is configurable by level and class name can be enabled for the execution service itself using a separate section in the `api.log4j.properties` file. It is located in `$SOAM_HOME/conf` (Linux) or `%SOAM_HOME%\conf` (Windows) on the host where the execution service runs.

Deploy executables

This topic describes the steps for deploying executables to compute hosts.

Create a deployment package

You must package your executable files before deploying them to Symphony.

1. Create the deployment package by compressing the executable files and any dependencies (for example, compress into a .zip or .gz file). The executable files should consist of the execution task's commands executable and pre-/post- commands, if applicable.

Edit an application profile

This procedure assumes that you are starting with a new application profile for execution application.

1. Open `SymexecApplicationProfileTemplate.xml` located in `%SOAM_HOME%\4.1\samples\Templates` (Windows) or `$SOAM_HOME/4.1/samples/Templates` (Linux).
2. Edit the application name, package name, and consumer name, if necessary.

   Important:

   Do not make other changes. The application profile must be configured for execution applications, i.e., the session types DetachableSession and UndetachableSession must both be defined in the application profile.
3. Save the file with a new name such as `SymexecApp.xml`.

Add an execution application to Symphony

You can add an application to Symphony by using the Add Application wizard in the Platform Management Console. The wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the wizard, your application should be ready to use.

1. In the Platform Management Console, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard.
5. Select your application profile xml file (for example, `SymexecApp.xml`), then click Continue

   The Service Package location window displays.
6. Browse to the created service package and select it, then, select Continue.

   The Confirmation window displays.
7. Review your selections, then select Confirm.

The window displays indicating progress. Your application is ready to use.

8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Retrieving results from execution tasks

The results from an execution task can be in the form of an exit code with optional stdout and stderr data files from the execution task's command.

Retrieving the exit code

Exit codes of execution tasks are stored in log files on the compute hosts where the execution tasks run.

There are various ways to retrieve the exit code:

- Log files can be retrieved through the Symphony Platform Management Console; refer to PMC help for more information. The logDirectory attribute in the application profile can be configured for log retrieval.

  Important:

  When configuring the application profile for log retrieval, you must not change the values of the fileNamePattern and subDirectoryPattern attributes. If these values are changed, the PMC will not be able to find the log files.
- If you run the execution application from the DE PMC, you can only retrieve the exit code by redirecting it to a file; refer to the next section on stdout and stderr for details.
- If you run the execution application using the Symphony client SDK, you can retrieve the exit code from `ExecutionStatus`.
- If you run the execution application using the symexec CLI, then the exit code will be printed out by symexec as the result for fetch and run options. You also can retrieve the exit code by looking at the execution session's log file.

Here is an example of an exit code entry in a log file:

```
2007-11-19 12:55:14.414905 hostname 3016 Session #3, Task #1
execution command: execution finished with exit code <0>.
```

Retrieving stdout and stderr data

To retrieve the stdout and stderr data from a completed execution task, you must change its command to run under a shell so the command can redirect the stdout and stderr data to file (s). The file(s) can then be retrieved by either copying the file from the remote host (for example, with an FTP command) or by using a shared file system location. The following examples demonstrate how to retrieve the stdout and stderr data from Windows and Linux hosts.

Windows:

In this example, the command will take the output of the "`dir` c:\" command and place it in the `dir_content.txt` file. The command can be entered in the Start > Run textbox. The **cmd /c** opens a command shell and closes it after processing the string.

cmd /c dir c:\ > c:\temp\dir_content.txt

Linux:

In this example, the command will take the output of the `"ls /"` command and place it in the `root_content.txt` file. The **sh -c** opens a command shell, which reads the commands from the string.

```
sh -c ’ls / > /tmp/root_content.txt’
```

CHAPTER

8

Development Guidelines and Best Practices

Client development guidelines

Type of program

A Symphony client does not need to be an end user program. It can be a middle-tier proxy server to pass multiple end users' compute requests to a grid, and then return the compute results back to the end users. It can also be a master application program or even a service that itself is a compute-intensive piece to work with many other services.

Uninitialization

Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

Serialization and deserialization

Remember to double-check that your `Message` object serialization and deserialization order are the same.

A client can call different service methods

The Symphony API only provides a basic service invocation mechanism via opaque input/output messages and an `onInvoke( )` service call. If needed, you can call different methods in a service on top of this mechanism.

Ensure permissions when writing logs

When you run a client, make sure client application has write permission under its current working directory since it needs to write logs, or change the directory in which the client application writes logs in the `api.log4j.properties` file.

Client applications log to the current directory by default. This is defined in the `$SOAM_HOME/conf/api.log4j.properties` on Linux, and `%SOAM_HOME%\conf\api.log4j.properties` on Windows.

The client log is updated once the client attempts to access the Symphony API. Ensure your client application has write privileges to the directory specified in the properties file.

Threads and multithreading

Synchronous clients are not required to do anything special or even be aware of the API's use of threads. Asynchronous clients need only to follow basic rules for working in a multi-threaded environment, such as for example, not blocking the callback thread. The API implementation for the most part hides its threading model from the developer.

Memory management in the client for Java and .NET

Client and service applications handle a lot of data, and out of memory errors may occur.

To help control memory, you can do the following in your code:

- When retrieving task output:
  - Retrieve task output asynchronously, using the Session Callback object.

  OR

  - Retrieve task output synchronously, in groups. For example, if you send 1000 large input messages, retrieve output in groups of 50 rather than all at once.
- On both the client and service, set unused references to null.
- When you have large data or many tasks, try to trigger the garbage collector periodically to collect unused objects and avoid running into memory issues.
- For applications that consume large amounts of memory, consider implementing Symphony serialization instead of native serialization. Use the following guidelines to determine memory requirements for processing byte arrays and strings.
  - Memory requirements for byte arrays:

    Symphony serialization—3x the size of the byte array

    Native serialization—4x the size of the byte array
  - Memory requirements for strings:

    Symphony serialization—2x the size of the string in bytes. For example, if a string has 10 characters = 20 bytes, it will require 40 bytes.

    Native serialization—2.5x the size of the string in bytes. For example, if a string has 10 characters = 20 bytes, it will require 50 bytes.

Recoverable clients

A recoverable client is a client that can tolerate an abnormal termination of its execution and is able to recover and continue to process workload. Recovery of such a client usually involves it being restarted and given enough context to allow it to connect and open an existing session that previously contained its workload. For this type of client, it is usually recommended to set the discardResultsOnDelivery attribute to “false” in the applicaton profile to allow for a simplified recovery procedure.

Large number of tasks

For large numbers of tasks, for example, if you have 100,000 tasks in one session, you can get better performance by retrieving output asynchronously in a callback function. If you prefer to retrieve output synchronously, it is recommended to retrieve output in smaller groups—for example, you get better performance if you retrieve output for 10,000 tasks at a time, instead of retrieving output for 100,000 tasks with one `fetchTaskOutput( )` call.

How many sessions to create

There are different ways to manage the Symphony session. You can:

- Close a done session immediately
- Keep the idle session open

Close a done session immediately

The simplest way is to create a session, tightly pack all the tasks in, get all the outputs out, then close the done session.

Keep an idle session open for quick responsiveness of loosely-packed tasks

For applications that have very short tasks and tasks that come in periodically, creating a session for every discrete task pack or every task is not efficient because a session is a heavier scheduling unit than a task. It takes longer to create and close a session than to send a task within an existing session.

For this type of application, create a session and keep it open even if you do not have tasks for a short time period. This way the system responds much faster.

When there is no task in a session, Symphony immediately moves the service instances from the idle session to other busy sessions.

It is worth noting that it is not a good idea to keep an idle session open for too long, because open sessions occupy system resources. As a best practice, close a session if it is idle for too long. (The appropriate length of idle time should be determined by the developer.)

Smart pointers

A smart pointer is an object that encapsulates a real reference. When an object is no longer required, a smart pointer frees it. As a developer, you need not be concerned about catching problems like memory leaks.

In Symphony, you use smart pointers for all objects that are not user-implemented. You never need to clean up an object that is created with the API. When objects are out of scope, they clean up themselves.

Remember:

Smart pointers do not exist for objects that are user-implemented such as service containers, messages, and common data. For these objects, you still need to free up memory and manage it.

Data

Limits on message size

Symphony has no hard limit for the message size other than the physical limits imposed by systems outside Symphony. These limits may be determined by the size of the physical and virtual memory, and the operating system. The maximum data size also depends on the type of serialization used. Here is a guideline:

Note:

Available memory refers to the usable physical memory at the moment, and not just the manufacturer's specification for the memory.

Windows limits

- Symphony Serialization: 500 MB maximum data size, due to the application memory limit of 2 GB for a 32-bit host; requires at least 2 GB of available memory on the host to support this.
- Native Serialization: 400 MB maximum data size, due to the application memory limit of 2 GB for a 32-bit host; requires at least 2 GB of available memory on the host to support this.

Linux limits

- Symphony Serialization: 800 MB maximum data size, provided the host has at least 3 GB of available memory.
- Native Serialization: 800 MB maximum data size, provided the host has at least 3.5 GB of available memory.

Optimum ratio of task message size to task compute time

The optimum ratio of task message size to task compute time depends on the network bandwidth and the performance target you want to achieve.

In a normal size grid (100Mbps or 1Gbps network, 500 CPUs), to achieve > 90% CPU efficiency, the best practices are:

- If the ratio of task data size/task compute time is less than 10KB/second, send the task data by value; otherwise send the task data by reference
- If the task data sending time is less than 10% of the task compute time, then send the task data by value; otherwise send the task data by reference

Distributing data among tasks: by value or by reference

A symphony session manager is responsible for managing and scheduling sessions, services, and tasks. Overloading a session manager with data is not a good idea because it slows task distribution across compute hosts. As a best practice, think about pass-by-value versus pass-by-reference.

Pass-by-value with sendTaskInput() to pass small, task-specific data

Use `sendTaskInput()` only to pass small amounts of task-specific data.

If the task-specific input and output data is small, a Symphony client or service can pass the input and output data by value. The client sends the data value in the Symphony task message through session manager. The service gets the data value through the Symphony message from session manager.

Pass-by-value with common data if the dataset resides in a client

If the shared market dataset resides in a client, the client can distribute the data with session common data. The data is distributed to the service instance when the service instance is assigned to the session. The service instance can access the common data from the `onSessionEnter()` method. The client can update the common data by using the update( ) method.

Service instances can cache the data in memory or the local disk for multiple tasks. You only need to use the `sendTaskInput()` call to pass small task-specific data.

Pass-by-reference with common data for large data or when dataset resides in a shared location

If the shared market dataset resides in a shared location such as a database, file system, or cache system, the client can distribute a reference to the shared data with session common data. The reference to the dataset is distributed to the service instance when the service instance is assigned to the session. The service instance can access the common data from the `onSessionEnter()` method. The client can update the common data by using the update( ) method..

Service instances can load the data from the shared location and cache it in memory or the local disk for multiple tasks.

You only need to use the `sendTaskInput()` call to pass small task-specific data.

Using external data sources

In addition to the session common data and task input/output data, the service instances and tasks can also receive input data from other data sources, and save output data to other data destinations. These data sources and destinations can be a database, a file server, a cache system, or even directly with the client application.

Data loss prevention

In a grid environment, there may be hundreds or thousands of compute hosts distributed in a cluster. In a typical risk management application, there may be hundreds of thousands of perturbations of market data/conditions. Each one of these can be a workload unit.

When you submit this workload to a grid, you expect the grid system to distribute the workload on grid, and guarantee processing without losing any workload, even if there are failures in hardware or software in:

- Grid management machines or software
- Compute machines and service applications

A reliable grid system should guarantee a transactional handling of application execution on the grid. A failure or even an entire system reboot should not require rerunning the workload from the beginning.

One problem in a traditional MPI-based parallel application is that when there is a failure in a distributed environment, the MPI-based application may fail and need to rerun from the beginning. Rerunning a large workload or the entire workload in the system not only wastes time and resources, but also may miss the time window of business opportunities.

Add recovery with recoverable sessions

Platform Symphony supports reliable computing by persisting Symphony session and task inputs and outputs. However, sometimes you may not want to recover your workload when a failure or error happens, or, you may want to trade persistency for performance— task persistency takes time and disk space and may slow down the overall system response time.

You can define whether a session is recoverable or non-recoverable in the application profile through the session type. In the client application, you can then specify the appropriate session type in createSession().

Choose a recoverable session when

- You have a long session that may last hours to compute many CPU-intensive tasks, and you do not want to waste CPU cycles to resubmit tasks in the session if a failure or error occurs.
- It is difficult or impossible to resubmit tasks in the session when a failure or error occurs.
- You have a mission-critical session that has to be finished before a deadline.

Choose a non-recoverable session when

- You have a short session that may only last for minutes, and you can always create a new session to resubmit tasks if a failure or error occurs.
- You want Symphony to immediately clean up the session and release the CPUs if a failure or error happens. Keeping this session running in the system is just waste of CPU cycles.
- You have an interactive online session that requires quick response time.

Implement application-level checkpointing for sessions

If you have long running tasks, you may not want to rerun a task from the beginning in case of failure.

A good practice is to have a long running task that periodically persists its intermediate results, such as every 10 minutes, so that when the task is rerun by Symphony, it can continue from where the last intermediate results that were persisted.

You need a persistent shared location like a persistent shared data cache or a shared file system because a task may be rerun on a different machine than previously.

Once a task can persist its intermediate results, you can perform application-level checkpointing by suspending the session.

A service instance can get an interrupt event by calling `serviceContext.getLastInterruptEvent()`, and use a grace period to persist intermediate results in a persistent shared location. Later on, either when the whole suspended session is resumed, or then the unfinished task is redispatched, another service instance picks up the task, and restores the intermediate results from the shared location.

Service development guidelines

Memory management

Any objects created by the API are managed by the API. You do not need to worry about memory. Use methods in your programming language to free memory for other objects.

How to control memory leaks

Forcing a service instance to restart is one way to control memory leaks.

About threads and multithreading

If you are using a service wrapper to execute UNIX commands and return results to the client, note that the service wrapper is implemented with fork/exec in the onInvoke() call. The onInvoke() call and other calls in Symphony are executed in a thread.

The fork/exec in a multithread environment causes race conditions that could result in deadlock. To resolve this, include -lpthread in the makefile so that the service uses the thread-aware fork in the pthread library instead of fork in glibc. If you are using a Makefile, change it to include the following:

```
LIBS = -L $(OUTPUT) -L $(TOP)/$(ARCH_BUILD)/lib \
-lsampleCommon -lsoambase -lsoamapi -lpthread
```

Library dependencies in clients and services

C++

When creating clients or services, pay special attention to library dependencies.

Compiling your client and service

Symphony API headers

Include `soam.h` in any source code that needs to refer to the C++ Symphony API.

Binary execution headers

Include `symexec.h` in any source code that needs to refer to the C++ Symphony Binary Execution API.

Location of headers

Symphony API and Binary Execution API header files are in:

- Windows: `%SOAM_HOME%\4.1\include`
- Linux/UNIX: `$SOAM_HOME/4.1/include`

Symphony API implementations

Include the C++ implementation file `SoamSrc.cpp` in any of your implementation files (.cpp files) so that all C++ Symphony API wrapper implementations can be compiled and linked to your client or service code .

Symphony binary execution implementations

Include the C++ implementation file `symexec.cpp` in any of your implementation files (.cpp files) to enable proper linking of the Binary Execution API wrapper implementation to your client.

Location of implementations

Symphony API implementations and Binary Execution API implementations are in:

- Windows: `%SOAM_HOME%\4.1\src`
- Linux/UNIX: `$SOAM_HOME/4.1/src`

If you are developing in Microsoft Visual Studio, you can include implementations directly into your project or from your `stdafx.cpp` file.

If you are developing in Linux/UNIX, you can include implementations in your Makefile instead of in the source.

Important:

Do not include implementation files in more than one place. Doing so introduces duplicate symbols and causes link failures.

Linking

On Windows:

- For Visual Studio compilers, use the /gr compiler option.
- Link to the static link library file `soambase.lib` to establish dynamic links to the `soambase.dll` library.
- This file is located in `%SOAM_HOME%\4.1\%EGO_MACHINE_TYPE%\lib`.
- For 64-bit applications, this file is located in %SOAM_HOME%\4.1\%EGO_MACHINE_TYPE%\lib64.

On Linux/UNIX:

- Do not disable the -rtti compiler option for gcc.
- Link to the `libsoambase.so` file in your Makefile.
- This file is located in `$SOAM_HOME/4.1/$EGO_MACHINE_TYPE/lib`.
- For 64-bit applications, this file is located in `$SOAM_HOME/4.1/$EGO_MACHINE_TYPE/lib64`.

Java

When creating clients or services, pay special attention to library dependencies.

Building your client and service

Symphony API libraries

The Java Symphony API contains two layers: a Java layer, and a C++ layer. The Java API is in the package `com.platform.symphony.soam`. Its Java implementation is located in `JavaSoamApi.jar`, and its C++ implementation is located in:

Windows:

- `jnativesoamapi_4.1.0.dll`(32-bit implementation)
- `jnativesoamapi_4.1.0_64.dll`(64-bit implementation)

Linux:

- `libjnativesoamapi_4.1.0.so`(32-bit implementation)
- `libjnativesoamapi_4.1.0_64.so`(64-bit implementation)

Important:

The jnativesoamapi_4.1.0.dll and jnativesoamapi_4.1.0_64.dll have implementation specific to Symphony 4.1 and cannot be separated from `JavaSoamApi.jar` in the same location.

Symphony binary execution libraries

The Symphony Java Binary Execution API is in the package `com.platform.symphony.symexec`. Its implementation is located in the `JavaSymexecApi.jar` file.

Location of libraries

Symphony API libraries and Binary Execution API libraries are in:

- Windows: `%SOAM_HOME%\4.1\%EGO_MACHINE_TYPE%\lib`
- Linux: `$SOAM_HOME/4.1/$EGO_MACHINE_TYPE/lib`

For 64-bit applications, you can use the same libraries.

C# .NET

Building your client and service

Symphony API assembly

The .NET API is in the namespace `Platform.Symphony.Soam`.

Its implementation is in the `Platform.Symphony.Soam.Net.dll` assembly. This assembly will be typically installed in the GAC to accomodate the execution of client and service from any location on a host on which Symphony is installed.

If you do not have administrative privileges during installation, you will need to manually install the assembly to the GAC . If manual installation to the GAC is not possible, you can copy the assembly into the same location as your client and deploy it in your service package as another dependent library of your service.

For 64-bit applications, use the `Platform.Symphony.Soam.Net_64.dll`.

Location of assembly

The `Platform.Symphony.Soam.Net.dll` assembly is in:

- `%SOAM_HOME%\4.1\%EGO_MACHINE_TYPE%\lib`

For 64-bit applications, the `Platform.Symphony.Soam.Net_64.dll` assembly is in:

- `%SOAM_HOME%\4.1\%EGO_MACHINE_TYPE%\lib64`

COM

Building your client

Symphony API libraries

The COM API contains implementation applicable only to clients.

To access the COM API, add a reference to the `Platform.Symphony.Soam.COM.dll` file in your project.

Location of libraries

The `Platform.Symphony.Soam.COM.dll` is in:

- `%SOAM_HOME%\4.1\%EGO_MACHINE_TYPE%\lib\COM`

CHAPTER

9

Symphony 64-bit Application Support

General considerations for porting existing applications to 64-bit

Symphony developers have the opportunity to deploy application logic compiled to run natively in 64-bit architectures. Note that if you have an existing application deployed in Symphony on a 32-bit architecture and plan to move it to a 64-bit platform, you should plan this activity carefully. Without proper planning, the resulting ported application could yield unexpected results that may delay rollout of the ported application.

Running in a 64-bit environment without porting

Even if your application is not ported to a 64-bit architecture, it may still be beneficial to run it in a 64-bit environment. This is because some 64-bit operating system implementations allow an application to allocate more memory than if it were run in the 32-bit version of the operating system.

An example is WindowsXP Pro 64, which allows application code to consume up to 4GB of memory as opposed to the 2GB allowed by its 32-bit implementation. Consult the operating system vendor to confirm this detail about your specific 64-bit operating system implementation.

Required changes to the application profile

If any of your compute hosts are 64-bit, you need to add an osType section in the Service sections with the type NTX64 or X86_64, depending on whether you have Windows or Linux hosts. If you do not specify the environment explicitly, then any is selected.

For example:

```
<Consumer applicationName="MyApplication" consumerId="/consumer"...
            ...
...
<Service name="MyService" description="Description of my service."
...
 <osType name="NTX64"
startCmd="${SOAM_DEPLOY_DIR}/MyService.exe"
workDir="${SOAM_DEPLOY_DIR}">
 </osType>
 <osType name="X86_64"
startCmd="${SOAM_DEPLOY_DIR}/MyService"
workDir="${SOAM_DEPLOY_DIR}">
            </osType>
 ...
</Service>
</Profile>
```

Additional information about 64-bit architectures

For more information about 64-bit architectures and the implication to your application, refer to public white papers that contain pertinent information on this topic. You can access such information from the following sources:

- Windows

  http://msdn2.microsoft.com/en-us/library/h2k70f3s(VS.80).aspx
- Linux

  http://www-128.ibm.com/developerworks/linux/library/l-port64.html
- Java

  http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html
- Building AMD64 Applications with the Microsoft Platform SDK

www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/30887.pdf

Considerations for porting existing C++ applications to 64-bit

Handling variables of type long and pointer

When porting your applications to the Linux 64-bit architecture, you must take special care when handling variables of type long and pointer, since the size of these variables may be platform dependent.

Windows

On Windows, 64-bit applications are supported natively within Visual Studio 2005.

- Refer to the Symphony C++ samples project settings in Visual Studio 2005 to see how to port your C++ application to 64-bit on Windows.
- When compiling the samples for 64-bit using Visual Studio 2005, use the solution files suffixed with "_vc80.sln". You must also select the "x64" configuration since by default the "Win32" configuration is selected by Visual Studio on first use.
- Note that only a "full" or "custom" installation of Visual Studio 2005 allows you to build 64-bit applications. A "typical" installation only installs the 32-bit components for building.

If you are not using Visual Studio 2005, the porting process involves installation and some configuration of the Microsoft Platform Software Development Kit. You can get details on this porting process from Microsoft's MSDN knowledge base.

Linux

On 64-bit versions of Linux, the GCC compiler defaults to compiling for the 64-bit architecture.

Considerations for porting existing Java applications to 64-bit

Pay special attention to the JVM that runs the application. The J2SE specification only added support for X86_64(AMD and Intel 64-bit architectures) version 1.5 of the JDK.

Considerations for porting existing .NET(C#) applications to 64-bit

Note:

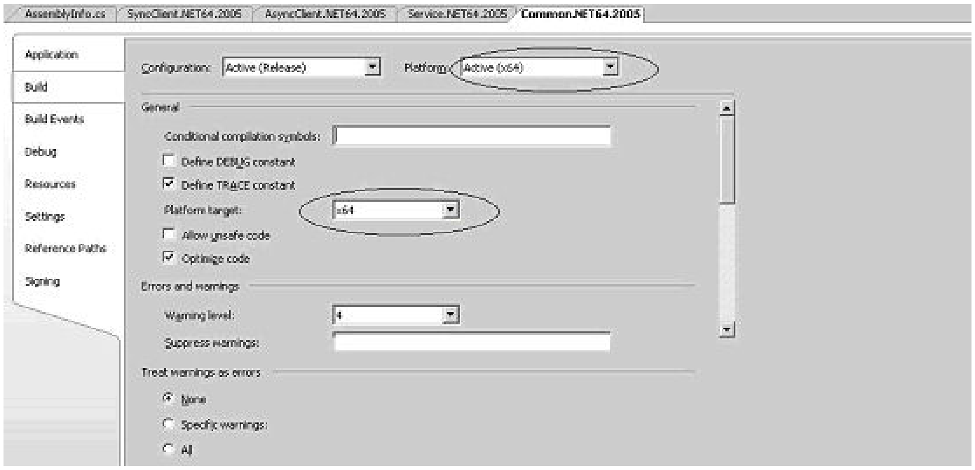
32-bit C# applications still work in a 64-bit environment.

- 64-bit C# applications can only run on .NET Framework 2.0 or above. When installing your development environment, you must ensure that either the 64-bit version of the .NET SDK or the full Visual Studio 2005 Professional package is installed on your 64-bit Windows host.
- You must add a reference to the 64-bit version of the .NET API (`Platform.Symphony.Soam.Net_64.dll`) in your Visual Studio 2005 application project files; see the 64-bit Symphony DE samples for details.
- Since 64-bit C# applications are only supported on .NET framework 2.0, you must use Visual Studio 2005 to build your 64-bit applications.

Compiling with Visual Studio 2005

When compiling the samples for a 64-bit environment using Visual Studio 2005, you must use the solution files suffixed with `64.2005.sln` or `64_2005.sln`.

1. In the Platform drop-down list, select x64. (By default, the x64 configuration is selected by Visual Studio on first use.)
2. In the Platform target drop-down list, select x64.



Configuration: Active (Release)
Platform: Active (x64)

Conditional compilation symbols:
☐ Define DEBUG constant
☑ Define TRACE constant
Platform target: x64
☐ Allow unsafe code
☑ Optimize code

Warning level: 4
Suppress warnings:

● None
○ Specific warnings:
○ All

PART

II

Debugging and Troubleshooting

CHAPTER

10

Debugging a Service

About debugging a service

There are four ways to debug a service:

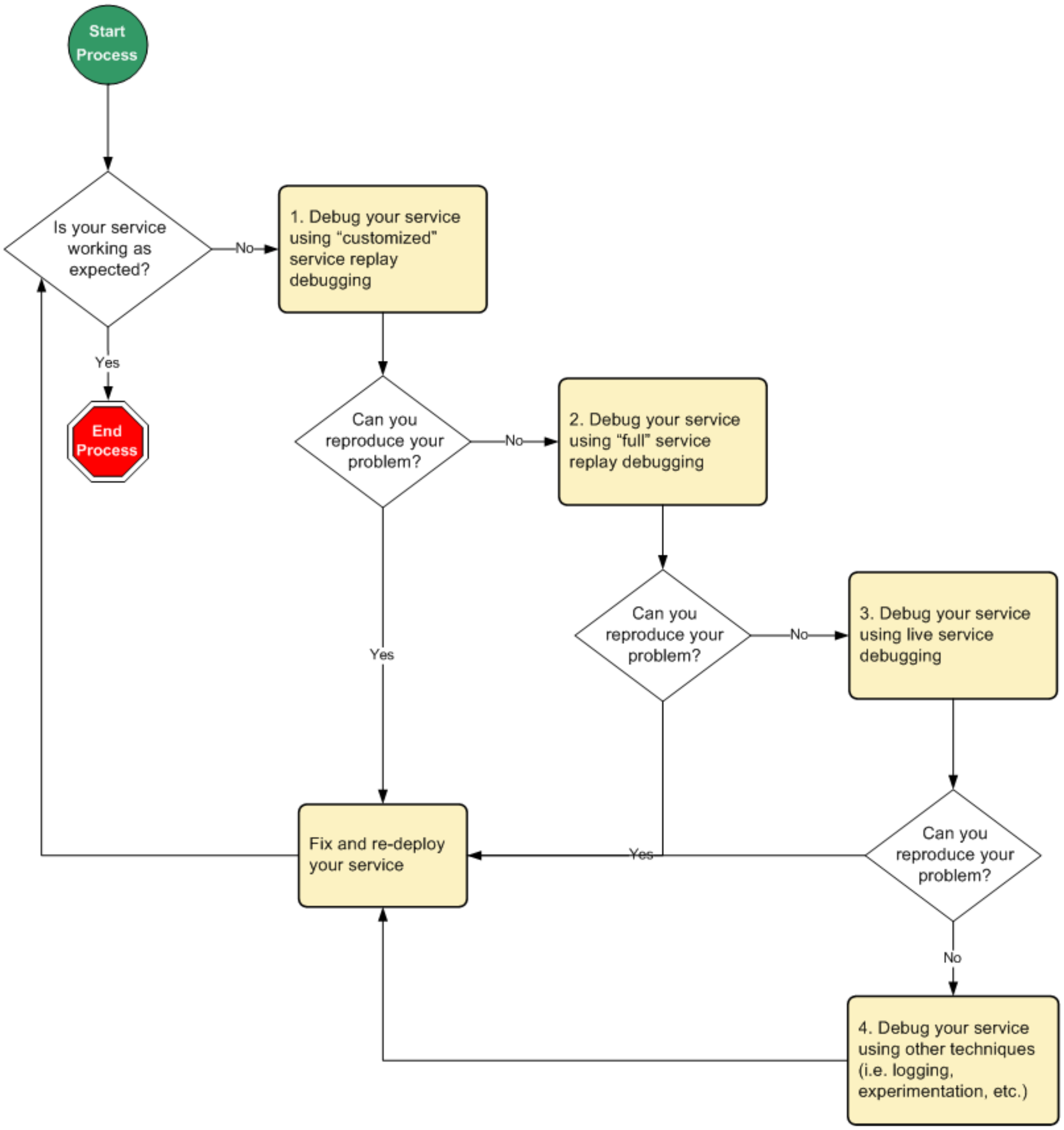
1. Customized service replay debugging
2. Full service replay debugging
3. Live service debugging
4. Ad hoc service debugging

Service replay debugging (customized or full) is best used if you need to test your code under *realistic* conditions:

- You do not want to limit the number of service instances (slots) available to your application,
- You do not want to modify the application configuration other than debugging configuration, and
- You do not want to modify your service code solely for debugging purposes.

The techniques discussed in this section apply to debugging a service in both Symphony DE and Symphony.

The following diagram describes the overall process to follow to debug a service.


Start Process
Is your service working as expected?
No
1. Debug your service using "customized" service replay debugging
Yes
End Process
Can you reproduce your problem?
No
2. Debug your service using "full" service replay debugging
Yes
Can you reproduce your problem?
No
3. Debug your service using live service debugging
Fix and re-deploy your service
Yes
Can you reproduce your problem?
No
4. Debug your service using other techniques (i.e. logging, experimentation, etc.)

Customized service replay debugging

Customized service replay debugging generates service event replay logs (SERLs) upon detection of any error-handling events you customize for that purpose. By default, customized service replay debugging generates files for the following common service problems:

- The service exits or crashes during the execution of a method,
- The service throws an exception during the execution of a method,
- The service times out while executing a method, and
- The service returns from a method with an exit code.

You can use the service event replay logs to try to reproduce your problem—you replay the relevant service events that occurred on the service instance.

You should be able to catch the most common service problems using this mode, particularly problems that are isolated (non-cumulative). If you cannot reproduce your problem in this mode, try debugging your service using the full service replay debugging.

Note:

Running your application in customized debug mode uses more memory on the compute host than running with no debugging, or running in full debug mode. You may not be able to use customized service replay debugging if you have large common data or large task input.

Tip:

Consider moving your application into production with customized service replay debugging enabled. You can test your service thoroughly in a grid environment until your service no longer produces service event replay logs in testing. Therefore any service replay debug logs generated in production reflect new problems (i.e. those that did not arise in testing).

Full service replay debugging

Full service replay debugging generates service event replay logs for every service instance in your cluster, regardless of whether an error occurs in that service instance.

You can use the service event replay logs to try to reproduce your problem—you replay the service events that occurred on the service instance.

You should be able to use this mode to catch problems that are cumulative. For example, this mode may help you to find that your service memory becomes more and more corrupted at each task invocation. You would also use full service replay debugging if your service problem does not generate a SERL in customized mode, or if you cannot reproduce the problem using customized service replay debugging.

Tip:

This mode is also useful for unit testing—stepping through your source code under a debugger during normal service execution.

If you cannot reproduce your problem using either customized or full service replay debugging, try debugging your service using live service debugging.

Live service debugging

Live service debugging allows you to debug your service instance as it is running live.

This mode gives you an exact picture of the runtime environment, not just a simulated environment.

If you still cannot reproduce your service problem using this mode, try debugging your service using ad hoc service debugging.

Ad hoc service debugging

If none of the above methods helps you resolve a service problem, try logging, experimentation, or other techniques you are familiar with.

Feature: Service replay debugging

The service replay debugging feature allows you to replay actual service events that occurred in your service instance when your application ran in Symphony. You use service replay debugging when you want to debug without making changes to your application.

About service replay debugging

In the Symphony environment, the SIM starts a service instance and drives all of the service events that occur. The service replay debugging feature can capture those service events in a file, along with the original data that was passed to the service at each stage. The service replay debugging feature allows you to run your service from this file, no longer requiring a SIM to drive the service actions. Service replay debugging can drive the service instance in complete isolation, independent of Symphony or Symphony DE.

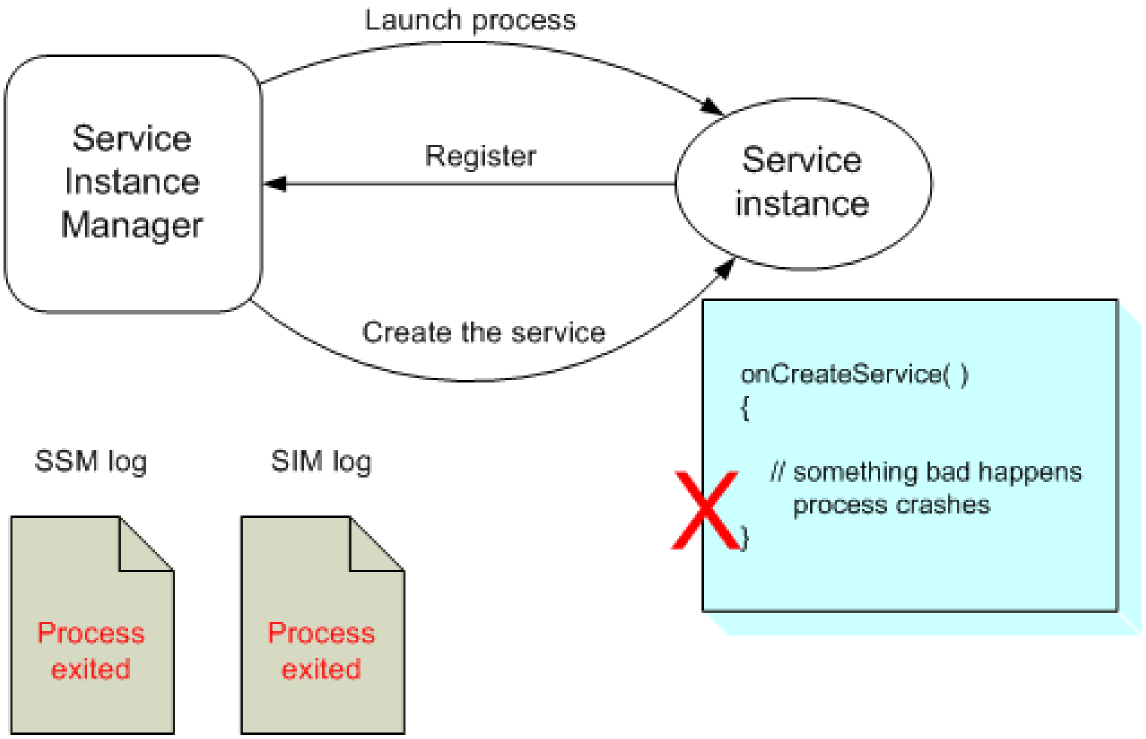
With service replay debugging, you can run your service binary directly from a command line, debugger, or integrated development environment (IDE) and step through the service code to find any errors in your service logic or in the service environment.

The service replay debugging feature makes it easy to debug a number of different scenarios:

To handle ...	The service replay debugging feature allows you to ...
Environment issues	Set the runtime environment for the service and run your service binary. In many cases, you will be able to see which dynamic libraries failed to load, which environment variables are not set as expected, and which other dependencies are missing.
Startup issues	Capture service startup and initialization issues without modifying your service code.
Miscellaneous issues	Capture every service event. For example, you could use the debugger to step through your service code to make sure everything works as expected.
Unexpected issues	Capture unexpected problems so that you can replay and debug them later on. For example, you could use service replay debugging to trace and resolve any new issues found in your service.

Without service replay debugging

The following is an example demonstrating debugging an error that occurs in the onCreateService( ) method when service replay debugging is not enabled.



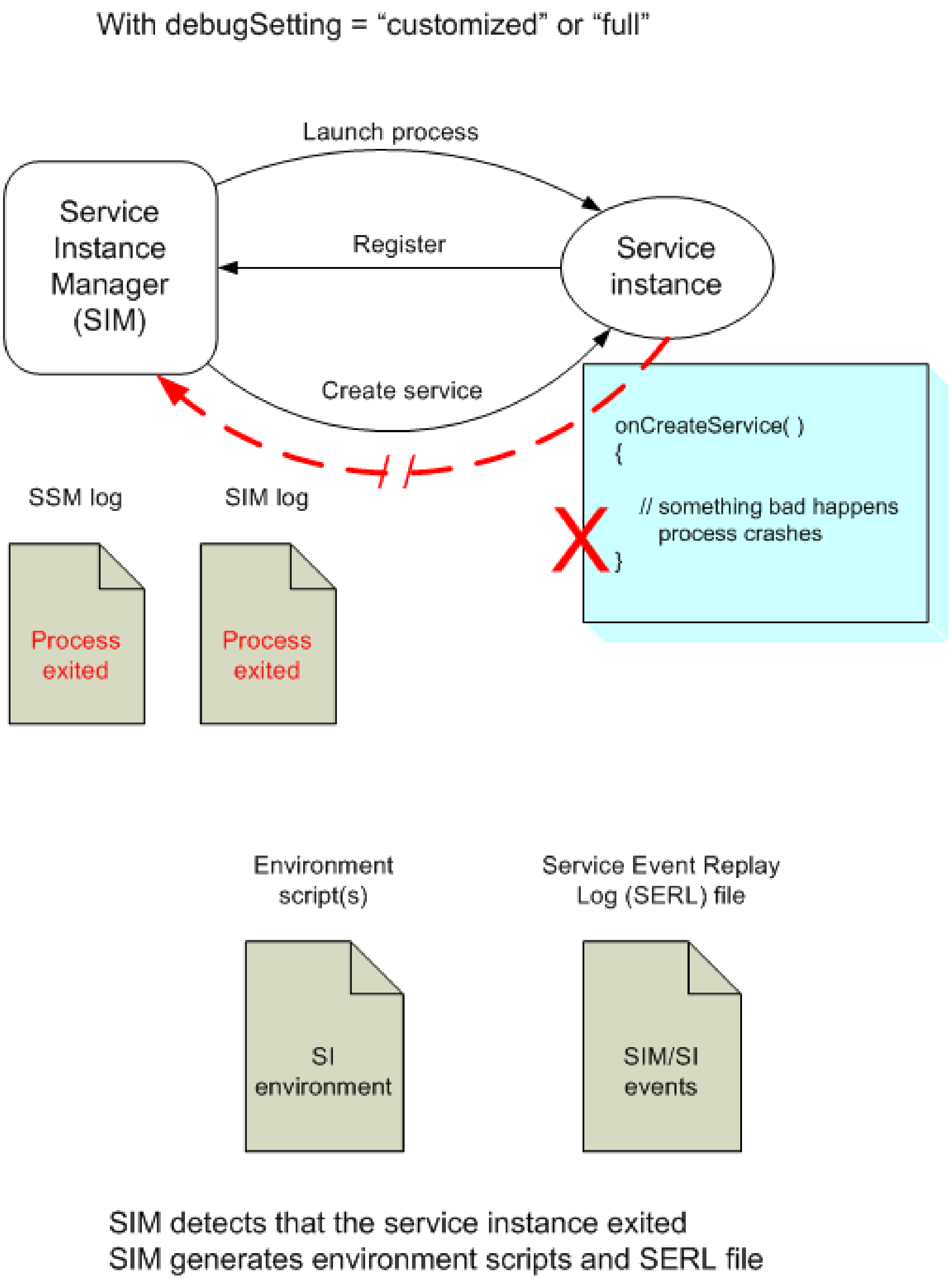
Without service replay debugging enabled, the session manager and service instance manager logs indicate that a process exited while creating the service. You do not know why.

To debug, you use live service debugging (debugging while the service is running), and you modify the onCreateService code to:

- Add an infinite while loop at the top of onCreateService
- Attach to the live service instance process with a debugger, and set breakpoints
- Modify the values of variables to exit the while loop
- Debug the live process
- Once resolved, remove the infinite while loop

With service replay debugging

The following is an example of debugging an error that occurs in the onCreateService( ) method when service replay debugging is enabled.



With debugSetting = “customized” or “full”

SIM detects that the service instance exited
SIM generates environment scripts and SERL file

To debug, you:

- Run the environment script in a command prompt or shell
- Run the service binary from the same command prompt or shell, as the environment script. Either run it directly or from a debugger IDE. The environment script tells the service to run from the SERL file.

Service event replay log (SERL)

This file contains the data required to replay actual service events that occurred on your service instance when your application ran in Symphony.

Note:

More than one set (a SERL file and an environment script) may exist, depending on the number of service instances that encountered errors. You can use any set to replay the service events that occurred.

Environment script

Environment scripts are used to simulate a service environment when you replay your service. When you run the environment script, the script sets the environment variables that were used when your service ran through Symphony.

An environment script is found together with its SERL file. The environment script is named as follows:

- Windows: `appName.serviceName.hostname.pid.timestamp.env.bat`
- UNIX bash: `appName.serviceName.hostname.pid.timestamp.env.profile`
- UNIX csh: `appName.serviceName.hostname.pid.timestamp.env.cshrc`

Symphony adds a single reserved environment variable in the environment script called SOAM_SERVICE_EVENT_REPLAY_LOG, which specifies the SERL file that can be used to run your service. When the SOAM_SERVICE_EVENT_REPLAY_LOG variable is defined by the environment script, your service is driven by the service events logged in the SERL file that this variable references. If you do not run the environment script, this environment variable is not defined, and your service is driven by the service instance manager.

Location of SERLs and environment scripts

The SERLs and environment scripts are located as follows:

- On Windows: `%SOAM_HOME%\work\serl\appName\serviceName`
- On UNIX: `$SOAM_HOME/work/serl/appName/serviceName`

Scope

Service replay debugging can be used in both Symphony DE and Symphony on the grid,

Service replay debugging can be used on all platforms supported by Symphony.

Configuration to enable service replay debugging

Service replay debugging is enabled by setting the debugSetting attribute in the Service section of the application profile.

When you set debugSetting to ...	The behavior is ...	Use this ...
none	Disables service replay debugging. No environment script or service event replay log is created. This is the default setting.	When you do not want to enable service replay debugging.
customized	Enables service replay debugging upon detection of specific, customizable error-handling events. Upon detection of a customized event, the environment in which the service is running is captured in one or more environment scripts, depending on the platform, and a service event replay log is created. You can configure the specific error-handling events that you would like Symphony to detect by specifying the customizedDebugAction for each of these events.	When you are developing and testing your service, to allow Symphony to detect common service problems, or when you are moving a clean service into production, and want to detect any unexpected problems that did not occur during testing.
full	Enables service replay debugging. For every service instance, the environment in which the service is running is captured in one or more environment scripts, depending on the platform, and a service event replay log is created capturing all service events.	When you cannot capture your problems using the customized debug setting, or when problems are cumulative.

Events captured when debugSetting is customized

For each method, the service events that are captured are listed below:

Method	Captured Service Events	Notes
Register	• Register	"Register" means "All code that executes before calling the ServiceContainer's run method".
CreateService	• Register • CreateService	
SessionEnter	• Register • CreateService • SessionEnter	
SessionUpdate	• Register • CreateService • SessionEnter • SessionUpdate	It is better to record all SessionUpdate events that had occurred between SessionEnter and SessionLeave. If invokes are interleaved between SessionUpdates, those Invoke events that occurred are discarded.

Method	Captured Service Events	Notes
Invoke	- Register - CreateService - SessionEnter - SessionUpdate - Invoke	If there were tasks that preceded the current task, they are not replayed. This assumes that the execution of those tasks did not contribute to the failure of the current task (i.e. the tasks do not alter the state of the service). Therefore, the omission of the other service events should not change the behavior of the Invoke when replayed. All SessionUpdate events that had occurred for this session are retained.
SessionLeave	- Register - CreateService - SessionEnter - SessionUpdate - SessionLeave	If there were tasks that preceded the SessionLeave, they are not replayed. This assumes that the execution of tasks do not contribute to the failure of the SessionLeave method (i.e. the tasks do not alter the state of the service). This also assumes that the SessionEnter is used correctly (user's intention for that function is correct, but the functionality may not be correct) to perform any session-specific initialization and that SessionLeave is used correctly to do the corresponding session-specific uninitialization. Therefore, the omission of the other service events should not change the behavior of the SessionLeave when replayed. All SessionUpdate events that had occurred for this session are retained.
DestroyService	- Register - CreateService - DestroyService	If there were sessions bound to this SI, tasks executed on this SI, and sessions unbound from this SI before the DestroyService, these service events are not replayed. This assumes that the CreateService is used correctly to perform any initialization common to the whole service and that the DestroyService is used correctly to do the corresponding service-specific uninitialization. Therefore, the omission of the other service events should not change the behavior of the DestroyService when replayed.
ServiceInterrupt		An interrupt event follows a "normal" service event (those listed above this one in this table). InterruptEvents are preserved only if the service event that precedes it is preserved for replay. Under service replay debugging, you have to control the interrupting thread and the thread that is performing the "normal" service event yourself to re-create whatever the scenarios you are experiencing. Symphony preserves service events but cannot preserve the timing.

Configuration to modify service replay debugging behavior

Service replay debugging behavior can be modified as follows:

- Customize the error-handling events you want to detect and capture when debugSetting is set to customized.
- Set the environment script execution mode to control the environment the service will replay in

Customize error-handling events for detection and capture

In the application profile, you can configure service replay debugging upon detection of the following error-handling events in a specific method:

- Timeout—A method executes for longer than its configured duration
- Exit—The service process exits while executing a method
- Exception—The service throws an exception while executing a method
- Return—The service returns from a method normally, possibly with a return code

You configure service replay debugging for an error-handling event by setting the customizedDebugAction attribute for the event.

Table 1: Effect of Setting customizedDebugAction

Event in Application Profile	customizedDebugAction Value	Behavior
Service > Control > Method > **Timeout**	writeServiceEventReplayFiles	Default. When Symphony detects that the specified method has timed out, it generates service event replay files to capture the relevant service events that led up to the timeout. This is the recommended setting if method timeout is an unexpected problem for your service.
	none	When Symphony detects that the specified method has timed out, it does not generate service event replay files. This is the recommended setting if your service hangs as a normal occurrence.
Service > Control > Method > **Exit**	writeServiceEventReplayFiles	Default. When Symphony detects that the service process has exited (or crashed) while executing the specified method, it generates service event replay files to capture the relevant service events that led up to the exit. This is the recommended setting if the service process exiting or crashing in the specified method is an unexpected problem for your service.
	none	When Symphony detects that the service process has exited (or crashed) during execution of the specified method, it does not generate service event replay files. This is the recommended setting if your service exits as a normal occurrence.
Service > Control > Method > **Exception**	writeServiceEventReplayFiles	Default. When Symphony detects that the specified method has thrown a particular exception (Fatal or Failure), as specified, it generates service event replay files to capture the relevant service events that lead up to the exception.
	none	When Symphony detects that the specified method has thrown a particular exception (Fatal or Failure), as specified, it does not generate service event replay files.

Event in Application Profile	customizedDebugAction Value	Behavior
Service > Control > Method > **Return**	writeServiceEventReplayFiles	When Symphony detects that the specified method has returned normally with or without a specific control code, it generates service event replay files.
	none	Default. When Symphony detects that the specified method has returned normally with or without a specific control code, it does not generate service event replay files. This is the recommended setting, as this is typically a normal, successful occurrence.

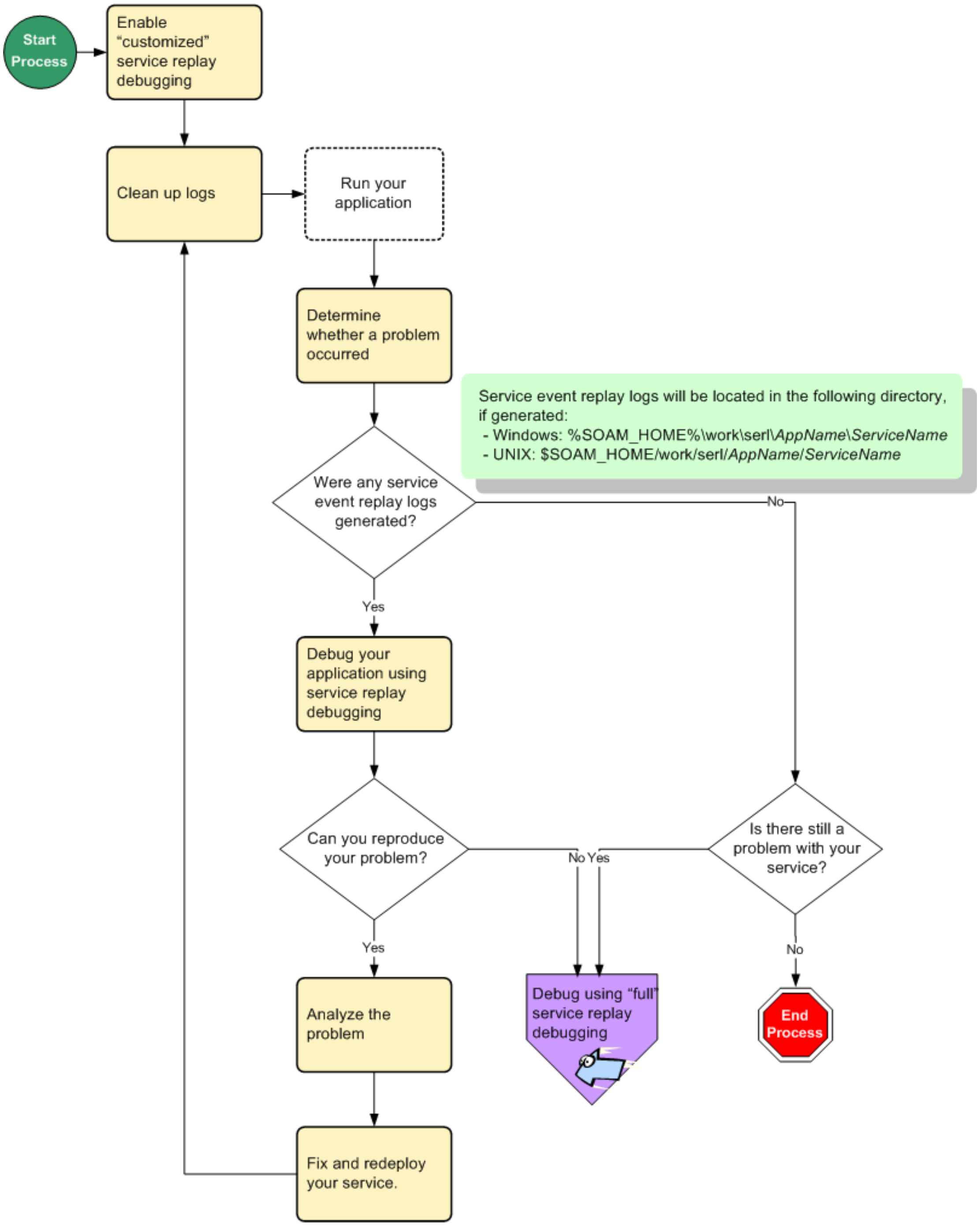
Environment script execution mode

The environment under which the service replays is captured in the environment scripts. Depending on the command-line arguments used when running the script, the environment is different.

If the run the script with ...	The behavior is ...	Use this mode when ...
No command-line arguments	The script appends the value of the current shell's environment to the service's environment for the following environment variables: • PATH • LD_LIBRARY_PATH (UNIX only) • LD_PRELOAD (UNIX only) Any other environment variables set in the service overwrite the values set in the shell.	You want to retain your development environment while debugging your service.
-pure-env (Windows) --pure-env (UNIX)	All environment variables set in the service overwrite the values set in the shell.	You cannot reproduce the problem using the default service environment, typically when you are trying to replicate the exact environment for the service to debug a possible environment problem.

Debug using customized service replay debugging

The following diagram provides an overview of the process you follow to debug a service.


Start Process
Enable "customized" service replay debugging
Clean up logs
Run your application
Determine whether a problem occurred
Service event replay logs will be located in the following directory, if generated:
- Windows: %SOAM_HOME%\work\serl\*AppName**ServiceName*
- UNIX: $SOAM_HOME/work/serl/*AppName*/*ServiceName*
Were any service event replay logs generated?
No
Yes
Debug your application using service replay debugging
Can you reproduce your problem?
No
Yes
Is there still a problem with your service?
Yes
No
Analyze the problem
Debug using "full" service replay debugging
End Process
Fix and redeploy your service.

Use customized service replay debugging

Enable customized service replay debugging.

1. In the console, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Click on your application in the table.

   The application profile editor displays.
3. Select Advanced Configuration. (The option may be currently set to Basic Configuration.)
4. Click Error Handling to expand the Error Handling section.
5. Select Customize event-driven - Write upon detection of events specified in table

   This causes the generation of a service event replay log file when any of the error-handling events in the table occur, provided the event is configured with Write as the customized debug action.

   The default configuration for customized debug actions generates a service event replay log file upon detection of any of the following error-handling events during the execution of a service handler method (for example CreateService, SessionEnter, Invoke, and so on):

   - The service throws FailureException or FatalException
   - The service exits or crashes
   - The method times out
6. Click Save to apply your changes.

Determine the problem

1. In the console, navigate to Symphony Workload > Monitor Workload. There are three types of errors to look for to quickly narrow down which compute host experienced service problems:

   - Service instance startup failures

     1. Look for your application in the application list.
     2. Look for service instance hostname and process ID pairs in the SI Startup Failures column for your application. If there are any service instance hostname and process pairs, the service application experienced problems in either the Register or CreateService method on each of the service instances listed in the column.

        If none of your service instances can start successfully, no service instance is available to run any of your tasks. If your client is hung waiting for task results, this may be the reason.
   - Binding failures

     1. Click on your application name to drill into more details about your workload.
     2. Look for service instance hostname and process ID pairs in the Binding Failures column for your session. If there are any service instance hostname and process ID pairs, the service application experienced problems in the SessionEnter method.
   - Task failures

   1. Navigate to the Sessions page. You can see whether any tasks ended in the ERROR state. It is possible your service experienced an error, but was unable to rerun and finish successfully.
   2. Click on a session to drill into more details about that session. Look for comments in the Failure Reason column of your Tasks table.
   3. Find out the host and process ID for the service instance on which the task ran. By default, the hostname is displayed in the Tasks table. You can configure the process ID to be displayed by modifying the Instance ID in Preferences at the bottom of the Tasks table.

   Make note of one hostname and process ID to troubleshoot a particular service instance to see what happened in the next steps.

2. Narrow down the detailed reason why service startup failed.

   In Symphony DE, look at the service instance manager logs for the application on the host for more information. For example:

```
%SOAM_HOME%\logs\sim.hostname.appName.log_file_number.log
```

   In Symphony, if you trap SNMP events, you receive event notifications for the service errors that occur.

   You can also use log retrieval to retrieve the service instance manager logs on the host where your service failed to start, as follows:

   a) Navigate to the Resources > Hosts (List View) tab in the console.
   b) Click on the host where service startup failures occurred.

      A dialog displays.
   c) Click on the Host Logs tab in the dialog.
   d) Check sim.log.
   e) Optional. Check User specified log or file if your service generates its own log files and you want to retrieve them.

      A text box appears.

      In the text box, type the file pattern to retrieve service logs for your application.
   f) Click on Retrieve Log List.

   From the log information, you can determine the following:

   - Which method caused the error (i.e. CreateService, Invoke, and so on)
   - Roughly what caused the error (process exit, FatalException, FailureException, unexpected exception, method timeout)
   - Where the service event replay log file is located

   The service instance manager reports something similar to the following example:

```
2007-11-14 12:30:43.843 Eastern Standard Time ERROR [3100:5188] sim.backend.ServiceBroker - Code
[S20070]: e:\symphonyde\de40\4.0\src\soamservicecontainer.cpp : 137 IException  Domain
<Application>: Unexpected service exception in method onCreateService(). If the additional details
attached do not provide enough information, change your code to use SoamException. Additional
Details: Unexpected Exception Caught in onCreateService()..
2007-11-14 12:30:43.843 Eastern Standard Time ERROR [3100:5188] soam.common.EventAgent - Code
[S75052]: Application <ServiceReplayDebuggerCPP>, service <DebugService>: Failure exception
thrown on method <createService>, control code <0>. Action on service instance: action <blockHost>
taken on service instance process <5672>, host <achin2>, service <DebugService>.
2007-11-14 12:30:43.890 Eastern Standard Time WARN [3100:5188] soam.common.EventAgent - Code
[S75064]: Application <FailureException with control code 0>: SIM detected <FailureException with
control code 0> for method <createService>.  Debug action is <writeServiceEventReplayFiles>. Serl
```

```
file is <E:\SymphonyDE\DE40\work\serl\ServiceReplayDebuggerCPP\DebugService
\ServiceReplayDebuggerCPP.DebugService.achin2.5672.2007-11-14.12h.30m.43s.859ms.serl>.
```

Debug your problem using service replay debugging

1. Open a command prompt or shell.

   Tip:

   On Windows, one way to open a command prompt is to select **Start** > **Run. . .** Type **cmd** and click **OK**.

   ---

2. Run the service replay debugging environment script in the command prompt to set the runtime environment for the service in the command prompt, as follows:

   - Windows: ***script_name*.bat**
   - UNIX csh: **source *script_name*.cshrc**
   - UNIX bash: **. *script_name*.profile**

   The environment scripts are located with the SERL files. For example, on Windows, in %SOAM_HOME%\work\serl\*appName**serviceName*.

   Tip:

   An easy way to set the environment is to copy the base file name from the service instance manager log that you were viewing in "Determining the problem". Change `.serl` in the file name to `.env.bat`, `.env.profile`, or `.env.cshrc` as required.

   On Windows, another easy way is to first navigate to the %SOAM_HOME%\work\serl\appName\serviceName directory using Windows Explorer, then drag the icon of the environment script into the command prompt, and press **Enter**.

   ---

3. Launch the debugger or IDE from the same command prompt or shell. The following examples demonstrate how:

   GDB:

   ```
   /usr/bin/gdb
   ```

   Visual Studio:

   ```
   devenv C:\mydirectory\mysolution_vc71.sln
   ```

   or, if your Visual Studio environment is not set in the command prompt:

   ```
   "C:\Program Files\Microsoft Visual Studio.NET 2003\Common7\IDE
   \devenv.com" \mydirectory\mysolution_vc71.sln
   ```

   Eclipse:

   ```
   C:\eclipse\eclipse.exe
   /usr/bin/eclipse/eclipse
   ```

4. Set breakpoints as appropriate.

   For example, if you already know that the problem is in CreateService, put a breakpoint at the beginning of the CreateService method. Otherwise, if you do not know where the problem is, put breakpoints at the beginning of each method.

5. Run your service application from the debugger or IDE and step through the code to find the problem.

   Tip:

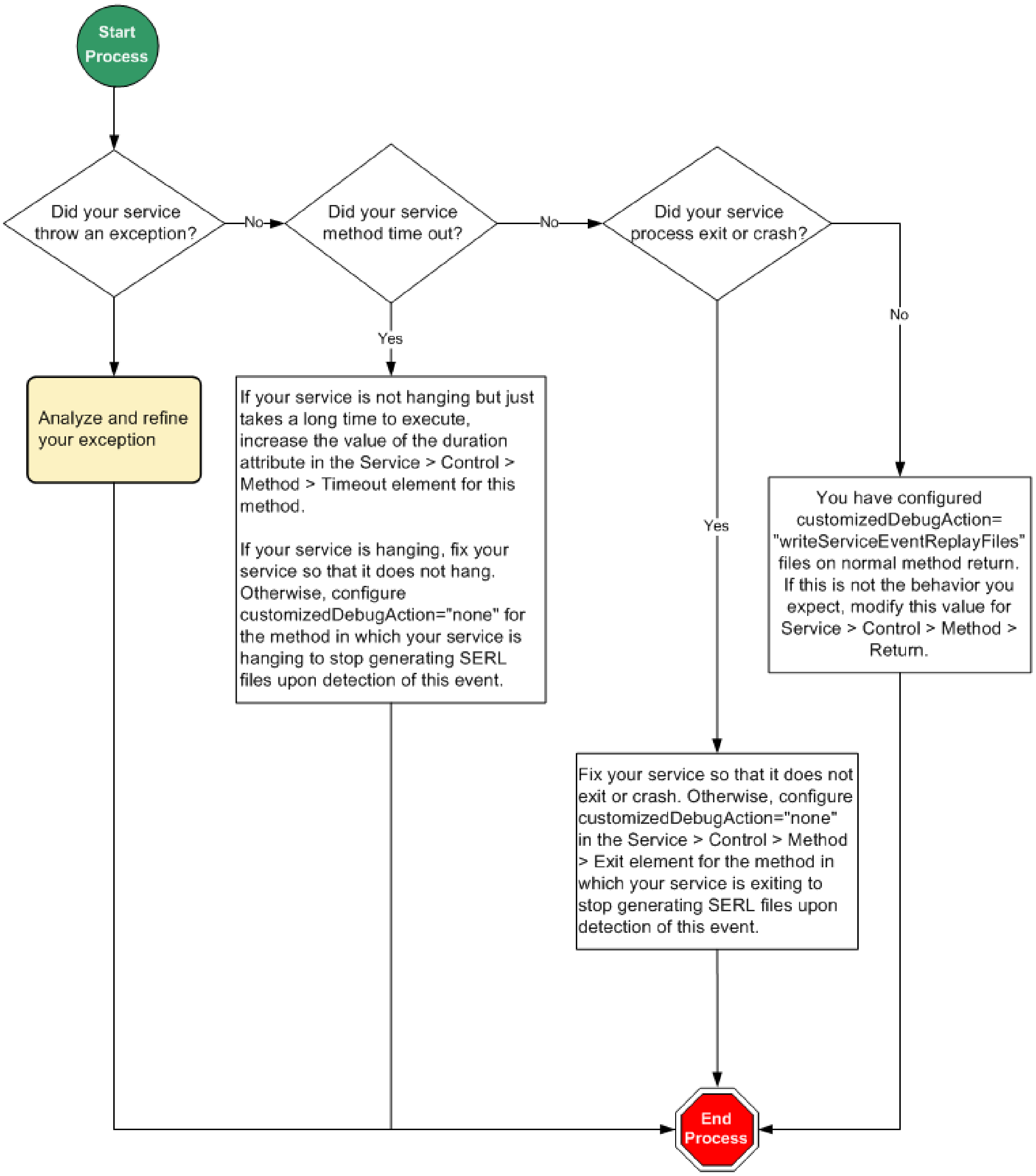
On Windows, remember to set the Service project as the Startup Project or Active Project before running your service application.

Note:

If there are multiple problems in a particular method, you should be able to catch them using one iteration of service replay debugging. If, however, there are problems in other methods that were not yet invoked when the service was run, no replay data exists for these methods. Actions cannot be replayed until they are played once through Symphony or Symphony DE.

Analyze the problem

1. Follow the process in the following diagram to help you analyze the problem.

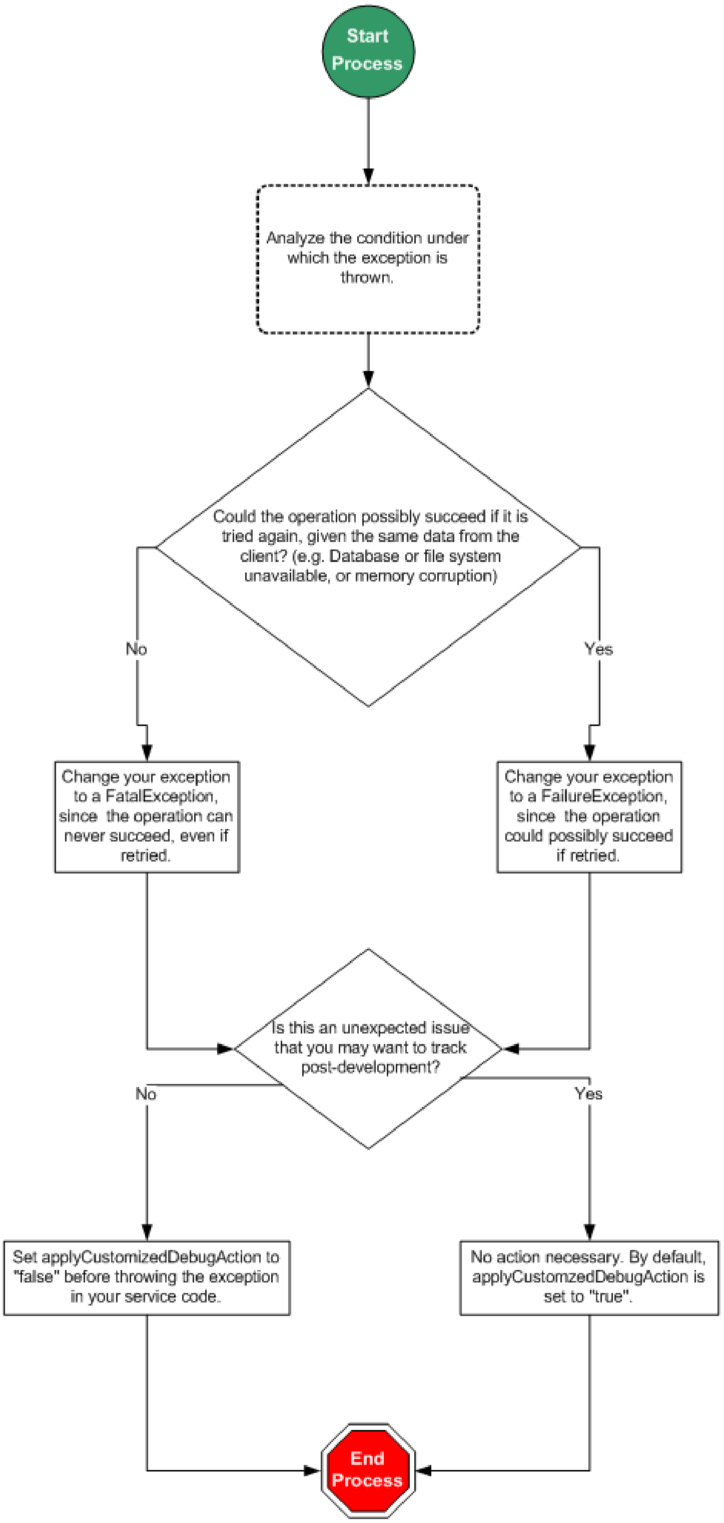

Start Process
Did your service throw an exception?
No
Did your service method time out?
No
Did your service process exit or crash?
No
Yes
Yes
Analyze and refine your exception
If your service is not hanging but just takes a long time to execute, increase the value of the duration attribute in the Service > Control > Method > Timeout element for this method.
If your service is hanging, fix your service so that it does not hang. Otherwise, configure customizedDebugAction="none" for the method in which your service is hanging to stop generating SERL files upon detection of this event.
You have configured customizedDebugAction= "writeServiceEventReplayFiles" files on normal method return. If this is not the behavior you expect, modify this value for Service > Control > Method > Return.
Fix your service so that it does not exit or crash. Otherwise, configure customizedDebugAction="none" in the Service > Control > Method > Exit element for the method in which your service is exiting to stop generating SERL files upon detection of this event.
End Process

Note:

To configure a customizedDebugAction, do the following:

1. Navigate to **Symphony Workload** > **Configure Applications**, and select your application from the list. The application profile editor displays.
2. Select **Advanced Configuration** from the drop-down menu at the top.
3. Under **Service Definition**, click on **Error Handling** to expand the section.
4. Select the method you want to modify, then select the customized debug action for the error-handling event you want to configure, and click **Save**.

The following diagram expands on the "Analyze and refine your exception"process in the diagram above.


Start Process
Analyze the condition under which the exception is thrown.
Could the operation possibly succeed if it is tried again, given the same data from the client? (e.g. Database or file system unavailable, or memory corruption)
No
Yes
Change your exception to a FatalException, since the operation can never succeed, even if retried.
Change your exception to a FailureException, since the operation could possibly succeed if retried.
Is this an unexpected issue that you may want to track post-development?
No
Yes
Set applyCustomizedDebugAction to "false" before throwing the exception in your service code.
No action necessary. By default, applyCustomzedDebugAction is set to "true".
End Process

For information on the applyCustomized DebugAction, see the API Reference.

Fix and redeploy the service

1. Make any necessary changes to your service code.
2. Rebuild the service.
3. Repackage the service.
4. Redeploy the service package.
   a) Navigate to Symphony Workload > Manage Service Packages in the console.
   b) Select Global Actions > Add package to repository.

      A dialog displays.
   c) Browse and select the updated service package.
   d) From the Select Application list, select your application (for this example, ServiceReplayDebuggerCPP).
   e) Click Add.

Clean up the logs

You clean up the service replay debugging environment scripts and SERL files at your discretion. You can remove individual files or the `serl` directory.

In production, depending on the frequency of error events, the number of SERL files may become unmanageable, requiring periodic cleanup.

1. Do one of the following:

   - For a small cluster, navigate to %SOAM_HOME%\work\serl\*appName**serviceName* directory, where the log files to clean up reside, and remove the desired files.

     Windows:

     `%SOAM_HOME%\work\serl\<appName>\<serviceName>`

     UNIX:

     `$SOAM_HOME/work/serl/<appName>/<serviceName>`
   - For a large Symphony cluster, you can use the `rfa` command to remove the distributed SERL directory for your application on a host-by-host basis. We recommend that you create a script to do this removal to save yourself manual work every time you want to remove the SERL files.

     For example, your script would contain the same command for each compute host in your cluster.

     Windows:

```
rfa remove -t hostA -s %SOAM_HOME%\work\serl\MyApp -p /MyConsumer -d
rfa remove -t hostB -s %SOAM_HOME%\work\serl\MyApp -p /MyConsumer -d
rfa remove -t hostC -s %SOAM_HOME%\work\serl\MyApp -p /MyConsumer -d
```

Linux:

```
rfa remove -t hostA -s $SOAM_HOME/work/serl/MyApp -p /MyConsumer -d
rfa remove -t hostB -s $SOAM_HOME/work/serl/MyApp -p /MyConsumer -d
rfa remove -t hostC -s $SOAM_HOME/work/serl/MyApp -p /MyConsumer -d
```

The options mean the following:

-t

The host on which to remove the file.

-s

The name of the file to remove.

-p

The consumer under which to execute the file removal. For the multi-user install, there is an OS execution user associated with each consumer.

-d

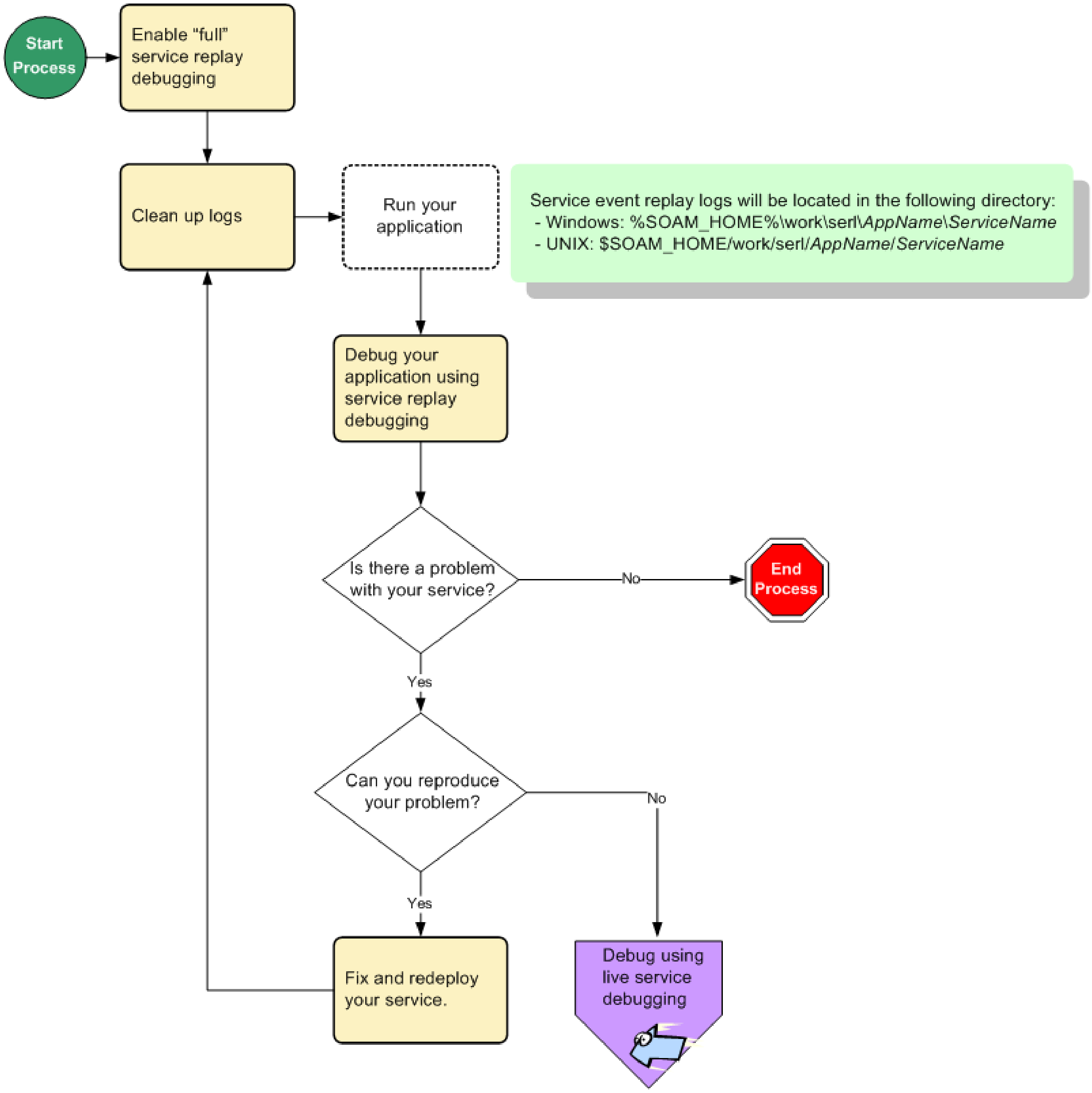
Flag that indicates that the specified file is a directory.

Run the client application

1. Run the client application to test the service.

Debug using full service replay debugging

Full service replay debugging generates service event replay logs for every service instance in your cluster, capturing all service events.


Start Process
Enable "full" service replay debugging
Clean up logs
Run your application
Service event replay logs will be located in the following directory:
- Windows: %SOAM_HOME%\work\serl\*AppName**ServiceName*
- UNIX: $SOAM_HOME/work/serl/*AppName*/*ServiceName*
Debug your application using service replay debugging
Is there a problem with your service?
No
End Process
Yes
Can you reproduce your problem?
No
Yes
Fix and redeploy your service.
Debug using live service debugging

Use full service replay debugging

Enable full service replay debugging

1. In the console, click Symphony Workload > Configure Applications.

   The Application page displays.
2. Click on your application in the table.

The application profile editor displays.

3. From the drop-down menu at the top of the editor, select Advanced Configuration. (It may be currently set to Basic Configuration.)
4. Click Error Handling to expand the Error Handling section.
5. From the drop-down menu, select Enable - Always write to debug file.

   This causes a service event replay log file to be generated for every service instance in this application with this service type.
6. Click Save to apply your changes.

Debug your problem using service replay debugging

1. Open a command prompt or shell.

   Tip:

   On Windows, one way to open a command prompt is to select **Start** > **Run. . .** Type **cmd** and click **OK**.
2. Set the run-time environment for the service in the command prompt by running the service replay debugging environment script in the command prompt. The environment script is named as follows:
   - Windows: `appName.serviceName.hostname.pid.timestamp.env.bat`
   - UNIX bash: `appName.serviceName.hostname.pid.timestamp.env.profile`
   - UNIX csh: `appName.serviceName.hostname.pid.timestamp.env.cshrc`
3. Launch the debugger or IDE from the same command prompt or shell. The following examples demonstrate how:

   GDB:

   ```
   /usr/bin/gdb
   ```

   Visual Studio:

   ```
   devenv.com mysolution_vc71.sln
   ```

   or, if your Visual Studio environment is not set in the command prompt:

   ```
   "C:\Program Files\Microsoft Visual Studio.NET 2003\Common7\IDE
   \devenv.com" mysolution_vc71.sln
   ```

   Eclipse:

   ```
   C:\eclipse\eclipse.exe
   /usr/bin/eclipse/eclipse
   ```
4. Set breakpoints as appropriate.

   For example, if you already know that the problem is in CreateService, put a breakpoint at the beginning of the CreateService method.Otherwise, if you do not know where the problem is, put breakpoints at the beginning of each method.
5. Run your service application from the debugger or IDE and step through the code to find the problem.

   Tip:

On Windows, remember to set the Service project as the Startup Project or Active Project before running your service application.

Note:

If there are multiple problems in a particular method, you should be able to catch them using one iteration of service replay debugging. If, however, there are problems in other methods that were not yet invoked when the service was run, no replay data exists for these methods. Actions cannot be replayed until they are played once through Symphony or Symphony DE.

Fix and redeploy the service

1. Make any necessary changes to your service code.
2. Rebuild the service.
3. Repackage the service.
4. Redeploy the service package.
   a) Navigate to Symphony Workload > Manage Service Packages in the console.
   b) Select Global Actions > Add package to repository.

      A dialog displays.
   c) Browse and select the updated service package.
   d) From the Select Application list, select your application (for this example, ServiceReplayDebuggerCPP).
   e) Click Add.

Clean up the logs

You clean up the service replay debugging environment scripts and SERL files at your discretion. You can remove individual files or the `serl` directory.

In production, depending on the frequency of error events, the number of SERL files may become unmanageable, requiring periodic cleanup.

1. Do one of the following:

   - For a small cluster, navigate to %SOAM_HOME%\work\serl\*appName**serviceName* directory, where the log files to clean up reside, and remove the desired files.

     Windows:

     `%SOAM_HOME%\work\serl\<appName>\<serviceName>`

     UNIX:

     `$SOAM_HOME/work/serl/<appName>/<serviceName>`
   - For a large Symphony cluster, you can use the `rfa` command to remove the distributed SERL directory for your application on a host-by-host basis. We recommend that you create a script to do this removal to save yourself manual work every time you want to remove the SERL files.

     For example, your script would contain the same command for each compute host in your cluster.

     Windows:

```
rfa remove -t hostA -s %SOAM_HOME%\work\serl\MyApp -p /MyConsumer -d
rfa remove -t hostB -s %SOAM_HOME%\work\serl\MyApp -p /MyConsumer -d
rfa remove -t hostC -s %SOAM_HOME%\work\serl\MyApp -p /MyConsumer -d
```

Linux:

```
rfa remove -t hostA -s $SOAM_HOME/work/serl/MyApp -p /MyConsumer -d
rfa remove -t hostB -s $SOAM_HOME/work/serl/MyApp -p /MyConsumer -d
rfa remove -t hostC -s $SOAM_HOME/work/serl/MyApp -p /MyConsumer -d
```

The options mean the following:

-t

The host on which to remove the file.

-s

The name of the file to remove.

-p

The consumer under which to execute the file removal. For the multi-user install, there is an OS execution user associated with each consumer.

-d

Flag that indicates that the specified file is a directory.

Run the client application

1. Run the client application to test the service.

Live service debugging

Live service debugging refers to debugging a service while it is running in its test or production environment in Symphony or in Symphony DE.

Use live service debugging if you have strict control over the environment:

- You can limit the number of service instances (slots) available to your application,
- You can control whether the application can be prestarted, and
- You can control when the workload is submitted.

Debug a service onSessionEnter( ), onSessionUpdate( ), onInvoke ( ), and onSessionLeave( )

1. Modify your existing application so that it creates and uses only a single session and sends only a single input to your service. If you want to debug the OnSessionUpdate( ) method, your client must submit an update to the service.

   The intent here is to remove all unnecessary interactions with the middleware so that you can isolate the problem and debug a single service instance without interference.

   Note:

   The system only calls `onSessionEnter()` and `onSessionLeave()` if common data is provided by the client. The system only calls `OnSessionUpdate()` if the client sends an update.

   The system invokes the `onSessionLeave(…)` method to unbind the current binding session when there is no more work for that session. Sending only one task ensures that the `onSessionLeave(…)` method is invoked shortly after the task completes.
2. Edit your application profile and change parameters as follows:
   a) Set the preStartApplication attribute to **true**.
   b) Set the numOfPreloadedServices attribute to **1**.

```
<Consumer applicationName="MyApplication" consumerId="/consumer" taskHighWaterMark="1.0"
taskLowWaterMark="0.0" preStartApplication="true" numOfPreloadedServices="1" />
```

3. Use the `soamreg` command or the Console to update your application with your new application profile.

   Your update causes the session director to immediately start a session manager to service your application (preStartApplication="true").

   Your application's session manager preloads a single service instance even before you send it any workload (numPreloadedServices="1"). The running service instance is not yet bound to a particular session.
4. Your service instance is a running instance of the executable that was produced by compiling your custom implementation of ServiceContainer. Attach a debugger to your service instance.
5. Now that you have attached to your service instance, set appropriate breakpoints in the `onSessionEnter(…)`, `onSessionUpdate(…)`, `onInvoke(…)`, and `onSessionLeave(…)` methods.
6. Run your client application.

   Provided that you modelled your client using the instructions in step 1, the middleware invokes the following methods on the service instance:

- `onSessionEnter(…)` once after the service is bound to your session, if there is common data
- `onSessionUpdate(…)` once per update after you have submitted the update from the client
- `onInvoke(…)` once (once per input sent to the service)
- `onSessionLeave(…)`once just before the service is unbound from your session

Debug a service onCreateService( )

If you simply put a breakpoint at the beginning of the CreateService method and then try to quickly attach a debugger to your service instance process, you will likely miss the opportunity to debug the code you were hoping to debug— the service instance manager initiates the invocation of the CreateService method almost immediately after it starts the service instance process, and the CreateService method may already execute before you attach your debugger to the service instance process.

To prevent the CreateService logic from executing before you are ready to debug it, add an infinite while loop to the beginning of the CreateService method. Make sure the infinite condition is not hard coded—you must be able to change the value of the variable to exit the while loop, so the execution flow can reach the true CreateService logic.

1. Modify your service to add the following loop to the beginning of your `onCreateService (…)` method:

```
// This is pseudo-code
boolean stall = true;
while (stall == true)
{
// Stall so machine is not bogged down by high CPU usage
 Sleep(1);
}
// Your original onCreateService(…) code
```

2. Compile and deploy your modified service.
3. Edit your application profile and change parameters as follows:
   a) Set the preStartApplication attribute to **true**.
   b) Set the numOfPreloadedServices attribute to **1**.

```
<Consumer applicationName="MyApplication" consumerId="/consumer" taskHighWaterMark="1.0"
taskLowWaterMark="0.0" preStartApplication="true" numOfPreloadedServices="1" />
```

4. Use the `soamreg` command or the Console to update your application with your new application profile.

   Your update causes the session director to immediately start a session manager to service your application (preStartApplication="true").

   Your application's session manager preloads a single service instance even before you send it any workload (numPreloadedServices="1"). However, because of the loop introduced in step 1, your `onCreateService(…)` method does not complete.
5. Your service instance is a running instance of the executable that was produced when you compiled your custom implementation of ServiceContainer. Attach a debugger to your service instance.
6. Set your breakpoints:

a) Add breakpoints that correspond to line 3 and line 8 in the pseudo-code above. When the debugger breaks at line 3, set the stall variable to **false**. This allows you to exit the infinite loop and step into your own code.
b) Continue. Your debugger breaks at line 8 to debug your original `onCreateService(…)` code.

Debug a service onDestroyService( )

If you simply put a breakpoint at the beginning of the DestroyService method and then try to quickly attach a debugger to your service instance process, you may miss the opportunity to debug the code you were hoping to debug. The DestroyService method may have already executed by the time you can attach your debugger to the service instance process. This is particularly likely if your workload completes very quickly, or if the service instance you want to debug starts up and shuts down very quickly if there is no workload to run.

To prevent the DestroyService logic from executing before you are ready to debug it, add an infinite while loop to the beginning of the DestroyService method. Make sure the infinite condition is not hard coded. You must be able to change the value of the variable to exit the while loop, so the execution flow can reach the true DestroyService logic.

1. Modify your service to add the following loop to the beginning of your `onDestroyService(…)` method:

```
// This is pseudo-code
boolean stall = true;
while (stall == true)
{
// Stall so machine is not bogged down by high CPU usage
Sleep(1);
}
// Your original onDestroyService(…) code
```

2. Compile and deploy your modified service.
3. Edit your application profile and change parameters as follows:
   a) Set the preStartApplication attribute to **false**.
   b) Set the taskLowWaterMark attribute to **1.0**.

```
<Consumer applicationName="MyApplication" consumerId="/consumer" taskHighWaterMark="1.0"
taskLowWaterMark="1.0"  preStartApplication="false" numOfPreloadedServices="1"/>
```

4. Use the `soamreg` command or the Console to update your application with your new application profile.

   Your update does not cause a session manager to be started because preStartApplication is set to false.
5. Run the client application.

   This causes a session manager and at least one service instance to be started.
6. Attach a debugger to the service instance:
   a) Your service instance is a running instance of the executable that was produced by compiling your custom implementation of ServiceContainer. Attach a debugger to your service instance.

b) Add breakpoints that correspond to line 3 and line 8 in the pseudo-code above. When the debugger breaks at line 2, set the stall variable to **false**. This allows you to exit the infinite loop and step into your own code.

c) Run your client application.

d) Continue. Your debugger breaks at line 8 to debug your original `onDestroyService (…)` code.

Retrieve application logs from the console

This is not applicable to Symphony DE.

Goal

You want to configure the system to be able to retrieve application logs that are written on all compute hosts on the grid from one central location, through the Console.

Note:

This feature is only available in Symphony, not Symphony DE.

Log location and naming

To retrieve logs through the Platform Management Console, you need to configure the location on which logs are stored on compute hosts.

For our example, this is what we want to configure:

- Services write logs in the following directory structure on compute hosts:

  ```
  ConsumerName/ApplicationName/SessionID
  ```
- Log files are named according to task ID, for example, `sampleService_task1.log`, `sampleService_task2.log`.
- The consumer name is /SampleApplications/SOASamples.
- The application name is sampleApp

Configure log location and naming using the command line

1. Open your application profile with an XML editor.

   Note:

   You can also use the Platform Management Console to change your application profile to configure parameters to log retrieval.
2. In the service section, define the `logDirectory`, `subDirectory`, and `fileNamePattern` parameters for each operating system type.

```
<Consumer applicationName="myApp"
...
<Service description="My own service" name="myService" deploymentTimeout="300"
packageName="myService">
        <osTypes>
           <osType name="NTX86" startCmd="${SOAM_DEPLOY_DIR}\myService.exe" logDirectory="$
{SOAM_HOME}\SampleApplications\SOASamples\myApp"  subDirectory="%sessionId%" fileNamePattern="_
%taskId%">
           </osType>
           <osType name="LINUX86" startCmd="${SOAM_DEPLOY_DIR}/myService" logDirectory="$
{SOAM_HOME}/SampleApplications/SOASamples/myApp"  subDirectory="%sessionId%" fileNamePattern="_
%taskId%">
       </osType>
        </osTypes>
</Service>
```

- **logDirectory**—Directory in which service log files for the application are to be logged. For our example, specify SampleApplications/SOASamples/myApp as the log directory.

  Important:

The path can be any desired path but, the path must be the same on all compute hosts.

   - **subDirectory**—Specify any subdirectory structure within the logging directory. For our example, the subdirectory is according to session ID. Since session IDs change depending on session, use the variable %sessionId%. Other supported variables in this parameter are %taskId%.
   - **fileNamePattern**—Specify how your files are named. The system matches part of the file name. In our example, the file name is `sampleservice_taskID.log`, so specify: %taskId% and the system will match the file names. Other supported variables in this parameter are %sessionId%.

3. Save your application profile.
4. Re-register your application profile to update your application profile in the system using the `soamreg` command or the console.
5. Run your client application so that the service can create logs in the log directories.

Configure log location and naming using the console

1. Navigate to Symphony Workload > Configure Applications.

   A list of applications displays.
2. Select your application from the list.

   The application profile editor dialog displays.
3. Select Advanced Configuration from the drop-down menu at the top of the dialog.
4. Click on Logging under the Service section to expand the Logging section.
5. Specify the Log directory.

   For our example, specify SampleApplications/SOASamples/myApp as the log directory.
6. Specify the subdirectory naming convention (if used).

   For our example, the subdirectory is according to session ID. Since session IDs change depending on session, use the variable %sessionId%. Other supported variables are %taskId %.
7. Specify the log file naming convention (if used).

   In our example, the file name is `sampleservice_taskID.log`, so specify: %taskId% and the system will match the file names. Other supported variables are %sessionId%.
8. Click Save to save your changes.

Ensure your service code writes to the configured location

1. Check your service code and ensure that services for your application write to the configured location and following the file name naming convention.

Retrieve application logs with the Platform Management Console

Download logs for an application

You can retrieve logs for an application that are created by the session manager of the application.

1. Find the application you want to retrieve logs for.

Click Symphony Workload > Monitor Workload > Applications. Expand the consumer tree on the left and click the leaf consumer that runs the application.

The application list displays with the Actions drop-down list on the right side.

2. Click Actions > Retrieve application system logs for the application you want.

   The log properties and log retrieval parameters display.

3. Select the logs you want from this host.Click Retrieve Log List.

   A list of logs displays.

4. Specify the volume of data:
   - Complete log
   - Number of lines
5. If you selected number of lines, enter the number of lines you want retrieved.

   The number of lines is counted from the latest event back through the log to the limit you specify.

6. Click the log file name.
7. Save the log file locally and open with any text editor.

Download logs for a task

You can retrieve the logs for a task that are created by your service application.

1. Click Symphony Workload > Monitor Workload.

   A list of applications displays.

2. Click the name of the application.

   A list of open sessions displays.

3. Click the session.

   A list of tasks displays.

4. Locate the task in the list.
5. For that task, select Actions > Retrieve Task Log.
6. Specify whether you want the log to be compressed or not.
7. Click the log file name.
8. Save the log file locally, uncompress if necessary, and open with any text editor.

Download logs for a service

Binding failures

You can retrieve the logs for a service in the event of a session binding failure in the service. The SSM lists up to five of the last binding failures.

1. Click Symphony Workload > Monitor Workload.

   A list of applications displays.

2. Click the name of the application.

   A list of sessions displays.

3. Consult the Binding Failures column for your session.

4. Click the SI host name that has logged the session binding failure.

   Host Logs page displays.
5. Select User specified log or file. Enter the pre-configured log file path.
6. Click Retrieve Log List.

   A list of log files displays.
7. Click the filename with the matching host name and pid of the service instance.
8. Save the log file locally, uncompress if necessary, and open with any text editor.

SI start-up failures

You can retrieve the logs for a service in the event of a service instance start-up failure. The SSM lists up to five of the last service start-up failures.

Note:

If a task fails (where a control code for task failure is set) or a session aborts, the SSM will list the host name and pid of the service instance.

1. Click Symphony Workload > Monitor Workload.

   A list of applications displays.
2. Consult the SI Startup Failures column for your application.
3. Click the SI host name that has logged the SI start-up failure.

   Host Logs page displays.
4. Select User specified log or file. Enter the pre-configured log file path.
5. Click Retrieve Log List.

   A list of log files displays.
6. Click the filename with the matching host name and pid of the service instance.
7. Save the log file locally, uncompress if necessary, and open with any text editor.

CHAPTER

11

Troubleshooting

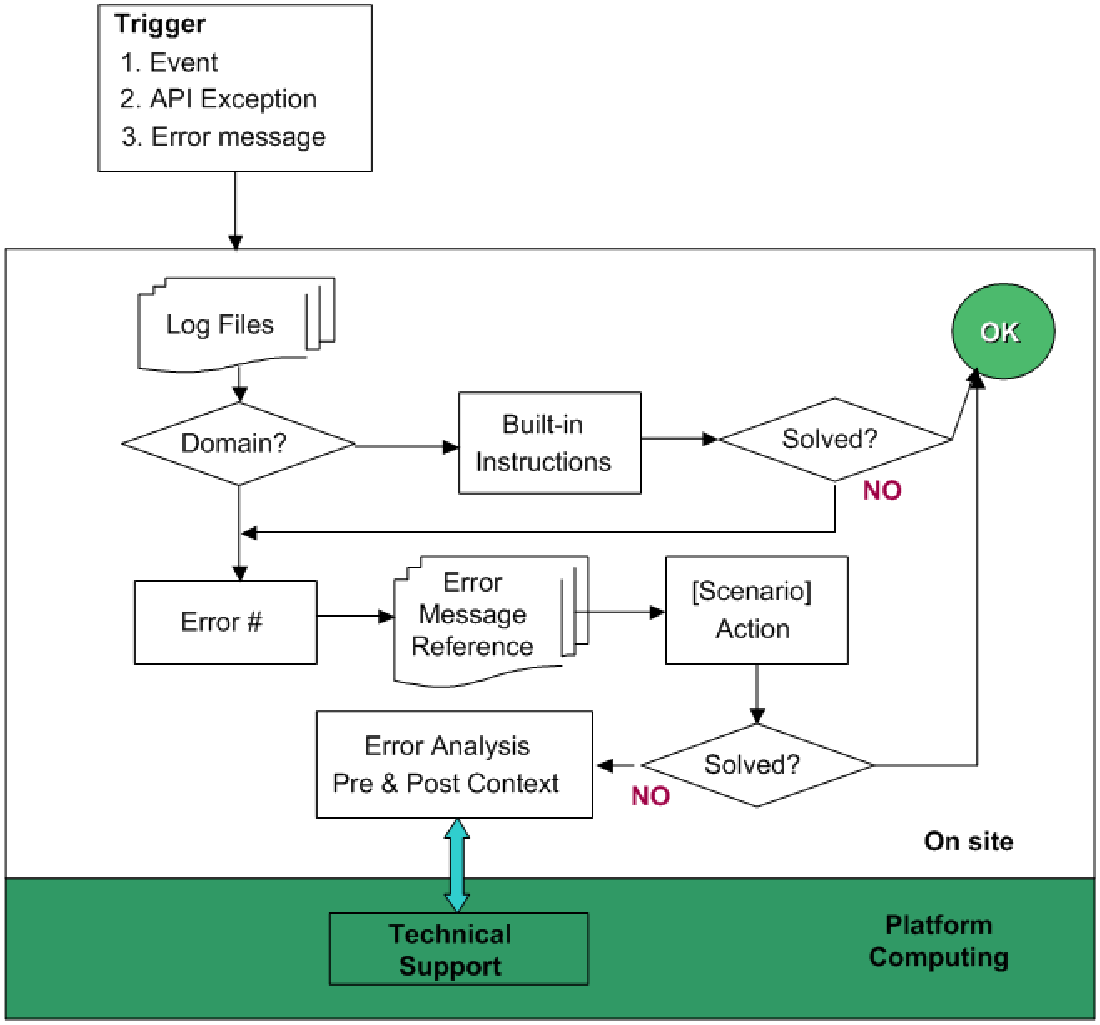
Troubleshooting overview

When problems occur in the system, you see them in the following ways:

- Error messages
- Events
- API exceptions

If the text in the error message does not provide enough information, check the *Error Message Reference*. This reference is installed with Symphony DE, and with Platform Symphony, and can be accessed from the knowledge centers.

Here is a flow chart illustrating how you can diagnose a problem:



Symphony events

Platform Symphony events can be monitored for and used to trigger actions automatically.

In Symphony DE, events are logged in the Symphony log files in the logs directory. There is no event framework.

In Symphony grid, you need to enable the event framework to be notified about events. In Symphony grid, EGO includes an SNMP plug-in that is integrated with SNMP, and uses SNMP traps as the notification mechanism. Only cluster administrators can enable the event framework. For more details, refer to the *Cluster and Application Management Guide*.

Platform Symphony events are categorized as follows:

- EGO system events, which identify host- and service-related occurrences within the cluster. Not available in Symphony DE.
- SOAM system events, which identify session-manager related occurrences within the cluster.
- Application events, which identify occurrences that affect workload.
- Platform Management Console events, which identify occurrences that affect the web server or the Platform Management Console itself.

SOAM system events

Event Name	Default level	Triggered when …
SYS_BM_BOUNDARY_BREACHED	Warning	The session manager memory usage exceeds threshold (%).
SYS_DS_READFAIL_SESSION SYS_DS_READFAIL_TASKINPUT SYS_DS_READFAIL_TASKOUTPUT	Error	The session manager failed to read from data storage.
SYS_DS_WRITEFAIL_SESSION SYS_DS_WRITEFAIL_TASKINPUT SYS_DS_WRITEFAIL_TASKOUTPUT SYS_DS_WRITEFAIL_SESSION_OBJECT SYS_DS_WRITEFAIL_TASK_OBJECT	Warning	The session manager failed to write to data storage.
SYS_FAILOVER_RETRIED	Info	Trying to restart the session manager or service instance manager.
SYS_SSM_DOWN	Info	The session manager goes down abnormally.
SYS_SSM_UP	Info	The session manager comes up.

Application events

Event Name	Default level	Triggered when …
SOA_SERVICE_BLOCKED	Error	A service instance is blocked from a host.

Event Name	Default level	Triggered when …
SOA_SERVICE_CUSTOM_ACTION	Error	A service instance returns a particular code.
SOA_SERVICE_DEPLOYMENT_FAILED	Error	A service failed to deploy.
SOA_SERVICE_EXITED	Error	A service instance exited.
SOA_SERVICE_FAILURE	Error	A service instance threw a failure exception.
SOA_SERVICE_FATAL_ERROR	Error	A service instance threw a fatal exception.
SOA_SERVICE_INIT_FAILED	Error	A service instance creation failed.
SOA_SERVICE_RUNAWAY	Error	A service instance takes longer than expected to complete.
SOA_SESSION_ABORTED	Error	A session is aborted.
SOA_SESSION_LOST	Error	A lost connection from the session is detected.
SOA_SESSION_PRI_CHANGED	Info	The priority of a session is changed and the session is resumed.
SOA_SESSION_RESUMED	Info	A session is resumed.
SOA_SESSION_SUSPENDED	Warning	A session is suspended.
SOA_TASK_EXIT	Error	A task exited, such as when a service instance crashes.
SOA_TASK_FAILURE	Error	A service instance threw a failure exception during the invoke call.
SOA_TASK_FATAL_ERROR	Error	A service instance threw a fatal exception during the invoke call.
SOA_TASK_RUNAWAY	Error	A task runs longer than expected and a timeout is invoked.

Platform Management Console events

Event name	Default level	Triggered when …
SYS_GUI_CPU_HI_WATER_MARK • Component name: GUI • Returned integer: 3	Warning	The web server host utilization exceeds the threshold set for `CPU_HIGH_MARK` in `wsm.conf`
SYS_GUI_MEMORY_HI_WATER_MARK • Component name: GUI • Returned integer: 2	Warning	The web server memory usage exceeds the threshold set for `MEM_HIGH_MARK` in `wsm.conf`

API exceptions

If your application code catches an exception, you can inspect the exception message to gain insight into the problem and decide possible actions to take.

If an exception is returned for a failed task result and if it contains an embedded exception, i.e., an exception from the service, you can inspect the error code and error message for the embedded exception to see what went wrong in the service; refer to *Error codes and embedded service exceptions* on page 96 for an example of how to retrieve an embedded exception. If there is no embedded exception, you can inspect the error message to determine the system reason for the failure. This information can be used to determine what the problem is and what actions to take.

About log files and levels

Use the Symphony log files to troubleshoot workload related components such as session director, session manager, and service instance manager.

Log files

The Symphony log files provide information on the general well-being of workload-related daemons and services.

Default log file locations

Symphony component log files

- Windows—`%`*`SOAM_HOME`*`%\logs`
- Linux/UNIX—`$`*`SOAM_HOME`*`/logs`

Symphony API log files

The Symphony API log file is written to the directory where the client executable resides.

Log file names

Log files are named according to the component they are logging and the host name where the component runs. For example, a log file for the session director running on hostA is named `sd.hostA.log`.

The following table lists possible log files and on which hosts they can be found.

Log file	Description	Host on which you can find the log file
sd.*host_name*.log	Messages, events, and errors for session director.	The host on which session director is running.
ssm.*host_name.application_name*.log	Messages, events, and errors related to workload scheduling for the specified application.	The host on which session manager is running.
sim.*host_name.application_name*.log	Messages, events, and errors related to tasks that ran for the specified application.	Each compute host running tasks for the application.
api.*host_name*.log	Messages, events, and errors for the client application that submits work to the system.	The host on which the client application runs.
agent.*host_name*.log	Only in Symphony DE. Messages, events, and errors related to startup and shut down of Symphony DE processes.	Only in Symphony DE. Found on every host on which Symphony DE runs.

Log file	Description	Host on which you can find the log file
cli.log	Messages, events, and errors related to the command line.	When enabled, found on the host from which the command was issued, in the directory from which the command was issued. By default, no `cli.log` files exist.

Logging configuration files

Default properties file location

The default locations of the logging configuration (properties) files are:

- Windows—`%`*`SOAM_HOME`*`%\conf`
- Linux/UNIX—`$`*`SOAM_HOME`*`/conf`

Available properties files

The following properties files are available:

- `agent.log4j.properties`
- `api.log4j.properties`
- `cli.log4j.properties`
- `rs.log4j.properties`
- `sd.log4j.properties`
- `sim.log4j.properties`
- `ssm.log4j.properties`

Log file formats

Log file entries follow a format that depends on the log level in which the message was logged.

File format customization

The format of the log-file entries can be changed. For more details, see the log4cxx documentation: `http://logging.apache.org/log4cxx/manual/classlog4cxx_1_1PatternLayout.html`

Synopsis for INFO log level

time_stamp log_level [*process_ID*:*thread_ID*] *logger_name* - *info_message*

Synopsis for WARN, ERROR, and FATAL log level messages

time_stamp log_level [*process_ID*:*thread_ID*] *logger_name* - Code[*Internal_Code*]: *file_name*:*line_number message*

Log file attributes

The following information is included for all messages recorded at the INFO, WARN, ERROR, and FATAL log levels:

```
time_stamp log_level [process_ID:thread_ID] logger_name
```

The following information is included for some errors:

```
Code[Internal_Code]: file_name:line_number
```

The remainder is the main body of the message. It can include information such as error domain, consumer ID, command, workDir, and hostname, along with the message.

Attributes of INFO, WARN, DEBUG, ERROR, and FATAL log level messages

time_stamp

Displays the time when the exception was thrown. The format for the time stamp is:

```
Year-month-day hour:minute:second.millisecond
```

Note:

For Linux/UNIX users only—By default, the time displayed in the logger files is GMT. The format of the timestamp can be changed by editing the related time zone settings in each `log4j.properties` file in $SOAM_HOME/conf. The properties files contain instructions on how to implement this change.

log_level

Displays the log level of the logger that logged the message.

Level	Description
FATAL	Logs only those messages in which the system is unusable.
ERROR	Logs only those messages that indicate error conditions or more serious messages.
WARN	Logs only those messages that are warnings or more serious messages. This is the default level of debug information.
INFO	Logs all informational messages and more serious messages.
DEBUG	Logs all debug-level and INFO messages.
ALL	Logs all available messages.

process_ID

Displays the ID of the Symphony component. The process ID is used to differentiate between daemons when more than one daemon of the same type runs on the host, such as when multiple session managers run on the same host.

Note:

The `soamview app` command displays the process ID of the session manager and `soamview task` displays the process ID of the service instance. The identity of the process that generated the message can be determined by comparing the process ID in the message with the process IDs displayed by `soamview app` and `soamview task`.

thread_ID

Displays the thread of the program that triggered the message.

logger_name

Displays the name of the logger component used to set the log level of the component that generated the message. The log4j logger components are listed in the properties files. These loggers are used to set the logging levels of specific components such as session director, session manager, service instance manager, repository service, and the CLI.

error_message

Displays the error message generated by the Symphony API.

error_code

Displays the error code that uniquely identifies the error. Error codes and their corresponding messages are listed in the *Error Message Reference*.

file_name

Displays the name of the source code file that triggered the message.

line_number

Displays the number of the line in the file that triggered the message.

domain

Displays the domain in which the message was triggered. Domains are virtual groups that categorize messages to precisely identify the component the message applies to.

Possible domains are:

- Application—Application configuration and deployment
- SOAM—Any Symphony component such as session manager and session director
- VEM—Resource management performed by EGO (not available in Symphony DE)
- OS—Operating system resource management of resources such as memory and disk capacity

PART

III

Application Deployment and Management

CHAPTER

12

Service Package Deployment

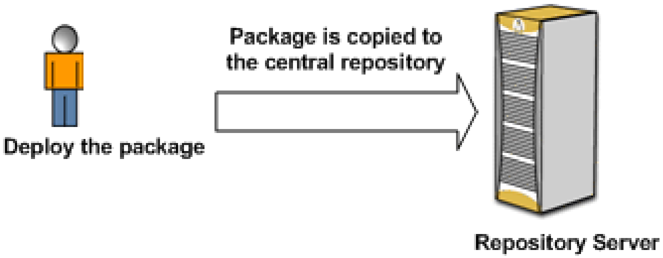
Service package deployment and removal process

The package deployment process has two phases: First, service packages are copied to the central repository on the repository server, the host on which the `rs` service is running. Then, when workload comes in, the service package is copied to compute hosts and uncompressed.

Package removal also has two phases: When a request to remove a package is made, service packages are removed from the central repository. Then, when a new application is deployed and existing packages on the compute hosts are no longer needed, packages are removed from compute hosts. For existing applications, when an existing package is updated, the packages that exist on compute hosts are overwritten when workload comes in.

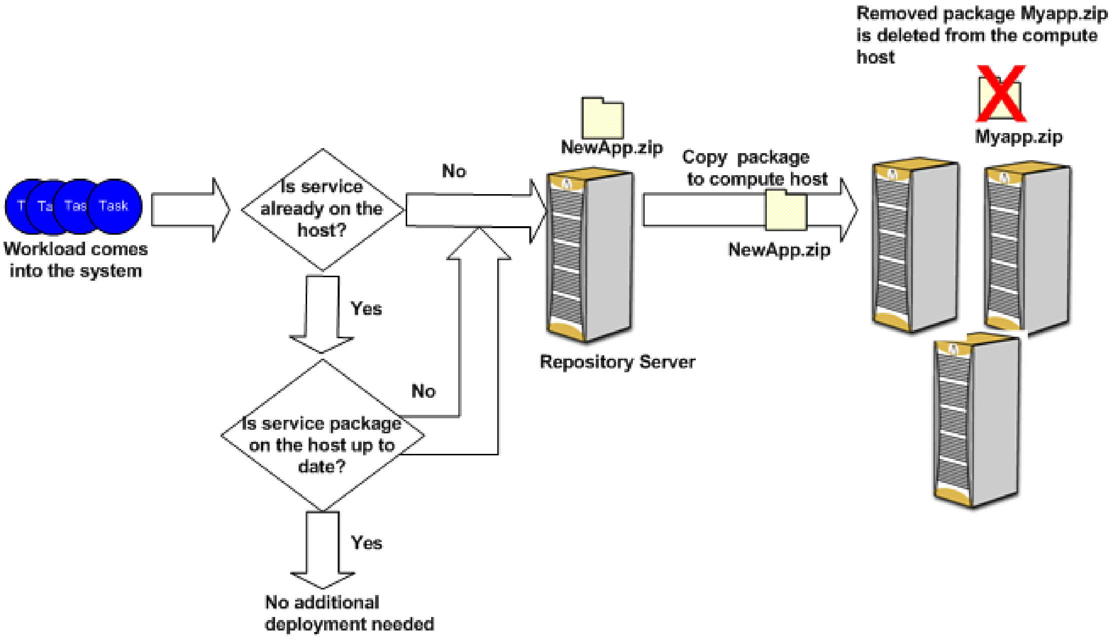
The package deployment process

1. You deploy the service package using the Platform Management Console or the `soamdeploy add` command. With the Platform Management Console, you use the Add/Remove Applications wizard or the global action Add package to repository in Manage Service Packages.
2. The package is copied to the repository server host.



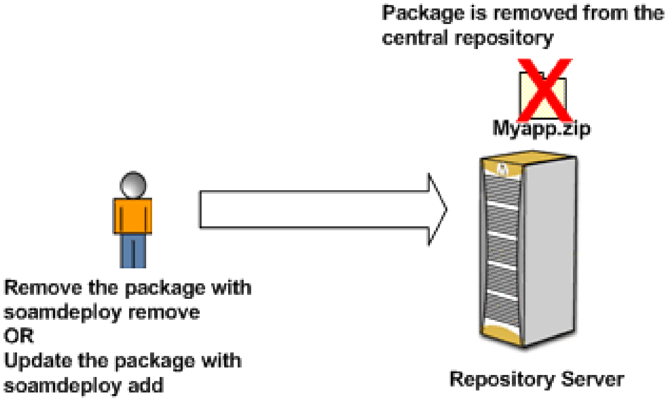
3. As workload comes in, the specified service in the application profile is requested for tasks. Platform Symphony checks whether the required service is already on the compute host.

   If the service is not already on the compute host, the Repository Service copies the service package from the repository server to the compute host, and uncompresses it, ready to be used.

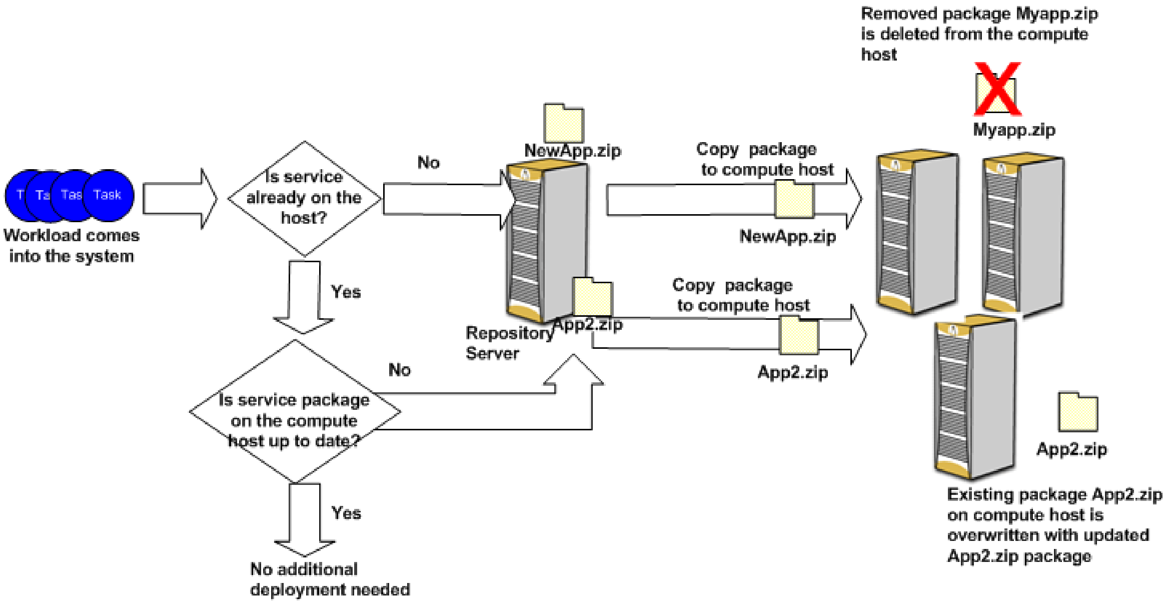

Removed package Myapp.zip is deleted from the compute host
Myapp.zip
T
Ta
Tas
Task
Workload comes into the system
Is service already on the host?
No
Yes
Is service package on the host up to date?
No
Yes
No additional deployment needed
NewApp.zip
Repository Server
Copy package to compute host
NewApp.zip

The package removal process

1. You request to remove the service package using the Platform Management Console or the `soamdeploy remove` command, or you update an existing package through the Management Console or the `soamdeploy add` command.
2. The package is removed from the repository server host.


Package is removed from the central repository
Myapp.zip
Remove the package with soamdeploy remove
OR
Update the package with soamdeploy add
Repository Server

3. Whenever a new package is deployed on to the host, the removed package is deleted. Whenever an updated package is deployed on to the host, the existing package is overwritten with the updated package.


Removed package Myapp.zip is deleted from the compute host
Myapp.zip
T Ta Tas Task
Workload comes into the system
Is service already on the host?
No
NewApp.zip
Copy package to compute host
NewApp.zip
Yes
Repository Server
App2.zip
Copy package to compute host
App2.zip
No
Is service package on the compute host up to date?
Yes
No additional deployment needed
App2.zip
Existing package App2.zip on compute host is overwritten with updated App2.zip package

Deploying a new application

Goal

You developed a new service, compiled it, and are now ready to use it in your cluster. To use the new service, you need to deploy it to compute hosts and associate it with an application.

At a glance

1. Create the service package
2. Create the application
3. Configure the client to run with the new application

Create the service package

Before you can deploy a service, you need to create a service package.

Packaging a Symphony application for deployment involves putting all service files and any dependent files associated with the service in a package.

Important:

Verify that all dependencies are either pre-installed or deployed with the service. For example, if your application is .NET, ensure that the .NET Framework is installed and that it is the correct version for your application.

Compress into a package:

- Service executables
- Additional files required for the services to work.

It is not required to use gzip as indicated in the example to package a service. You can use any supported format. If using a utility other than gzip, ensure the compression and uncompression utility is in your Path environment variable when using soamdeploy.

Supported package formats:

- .zip
- .tar
- .taz
- .tar.zip
- .tar.Z
- .tar.gz
- .tgz
- .jar
- .gz

On Windows

1. Go to the directory in which the service is located.

   For example, `%SOAM_HOME%\4.1\win32-vc7\samples`
2. Create an application package by compressing the service executable into a zip file:

   gzip SampleServiceCPP.exe

You have now created your first service package `SampleServiceCPP.exe.gz`. Next, create the application.

On Linux

1. Go to the directory in which the service is located.

   For example, `$SOAM_HOME/4.1/linux2.4-glibc2.2-x86/samples`
2. Create an application package by compressing the service executable into a tar file:

   tar -cvf SampleServiceCPP.tar SampleServiceCPP

   gzip SampleServiceCPP.tar

   You have now created your first service package `SampleServiceCPP.tar`.gz. Next, create the application.

Create the application

Add the application with the Add Application Wizard. After completing the Wizard, your application should be ready to use.

In Symphony DE, the Wizard deploys your service package and registers your application. In Symphony grid, the Wizard in addition creates a consumer with your application name and allocates resources,

You can also use commands to deploy a service package and register an application. See `soamdeploy` and `soamreg` in the *Symphony Reference* for more details.

1. In the Platform Management Console, click Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Create new profile and add application wizard.
5. Enter your application name, then click Continue.

   The Define the Service window displays.
6. Enter service information, then click Continue.
   a) Change the Service name to the name you want to assign to your new service.
   b) In command to start this service, enter the command to run your service executable.

      For example, if in your service package you have the directory structure `\myservice \myservice.exe`, indicate: `${SOAM_DEPLOY_DIR}/myservice/myservice`

   The Define session type window displays.
7. If you have defined session types in your client application, select Define a custom session type, fill in the desired information, then select Continue.

   The Confirm application profile details window displays.
8. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.

9. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Configure the client to run with a new application

Ensure client parameters match application profile parameters.

You created an application profile, indicated a service in the profile, and deployed the service package. You now want to use your new application with a client.

1. Check the client code to ensure the application name specified in the connection is the same as that specified in the application profile.

   For example, if your application name is `myapp`, your client code must also contain `myapp`:

```
...
// set up application specific information to be supplied to the System
char * appName = "myapp";...
...
```

2. Ensure the session type in the client code matches that specified in the application profile.

   For example, if in your application profile you have the session type ShortRunningTasks, your client code must also specify the same session type:

```
...
// Create a synchronous Session        SessionPtr sesPtr = conPtr->createSession("mySession",
"ShortRunningTasks",SF_RECEIVE_SYNC);
...
```

Note:

If you have not selected to create a custom session type and used the default session type, specify DefaultSession as the type.

3. Compile your client and run it on any host in the cluster.

Deploy a service package

To use a new application, you must deploy the service binary to your cluster. Using the `soamdeploy add` command to update an existing package will not terminate the current running workload. You can also deploy the service package to a non-leaf consumer, so that all applications registered to child leaf consumers are able to share the same service package.

Deploy a Windows service package

Verify that all dependencies are either pre-installed or deployed with the service. For example, if your application is .NET, ensure that the .NET Framework is installed and that it is the correct version for your application.

To use a new application, you must deploy the binary to hosts in your cluster.

1. Deploy the service package with the `soamdeploy` command:

   ```
   soamdeploy add SampleService -p SampleService.exe.gz -c /SampleApplications/SOASamples
   ```

   The service package is deployed.
2. Check the list of deployed services with the `soamdeploy view` command:

   ```
   soamdeploy view -c /SampleApplications/SOASamples
   ```

Deploy a Linux/UNIX service package

1. Deploy the service package with the `soamdeploy` command.

   ```
   soamdeploy add SampleService -p SampleService.tar.gz -c /SampleApplications/SOASamples
   ```

   The service package is deployed.
2. Check the list of deployed services with the `soamdeploy view` command:

   ```
   soamdeploy view -c /SampleApplications/SOASamples
   ```

Register a new application

You have to register a new application to start using it.

Tip:

As an alternative, you can use the Add/Remove Application wizard, a tool that walks you through all the steps required to successfully add an application to your cluster. Access the wizard from the Symphony Workload page, **Configure Applications** > **Add/Remove Application** or from the Start menu on Windows.

---

1. Click Symphony Workload > Configure Applications.

   A list of enabled and disabled applications displays.
2. Select Global Actions > Add Application using the profile editor.

   The Register a new application window displays.
3. From the drop down list, select Basic Configuration or Advanced Configuration.
4. Fill in the values.
5. Click Register.

Remove an application

You want to remove application binaries from the system.

When you remove an application through the Platform Management Console, the application is unregistered, the associated service package removed from the repository, and in Symphony grid, the associated consumer deleted.

Unregistering the application:

- Terminates existing sessions and tasks
- Releases all resources allocated to the application
- Unregisters the application
- Removes the service package from the repository if it is not shared with any other application(s)

Note:

You can also use the `soamunreg` command to unregister an application, and `soamdeploy remove` to remove the service package. For example:

```
soamunreg SampleAppCPP
```

```
soamdeploy remove SampleService -c /SampleApplications/SOASamples
```

1. In the Platform Management Console, click Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Remove an existing application, then Continue.

   The Remove an application page displays.
4. Follow the prompts.

Deploy a service package with your own deployment tool

You do not want to use the deployment tool distributed with Symphony to deploy your service packages. You have your own tool but want it to work with Symphony. You do not need to create a service package to use your own deployment tool.

- If the service binaries are in a shared location, service binaries and any other additional files required by the service must be accessible to all compute hosts
- If service executables are locally installed on compute hosts, the service executables must be in the same location on all compute hosts
- Symphony grid only. The OS user account assigned to the consumer for the application must have permissions to execute the service binaries on compute hosts

1. In the Platform Management Console, create an application with the Add/Remove Application wizard.
2. Click Configure Applications and select the application to modify.

   The Application Profile window displays.
3. Click Export and save it to a file.
4. In an XML editor, open the application profile and edit the Service section:
   a) For PreExecCmd , specify the command to run for your deployment tool to deploy your service on to the compute hosts.
   b) For StartCmd , specify the location of the service binary on the compute host after the deployment command has run. This location must be the same on all compute hosts.
   c) In workDir, specify the working directory for your service.

      On Windows:

      For example, if your deployment command is called `deploy`:

```
<Service name="myservice" description="My Sample Service">
...
<osType name="all" preExecCmd="C:\mydeploytool\bin\deploy.exe download -a myservice" startCmd="C:
\myservices\myservice\myservice.exe"
workDir="C:\myservices\myservice\work">
</osType>
...
</Service>
```

If working directory is not specified, by default, `%SOAM_HOME%\work` is used for service instances.

On Linux/UNIX:

```
<Service name="myservice" description="My Sample Service">
...
<osType name="all" preExecCmd="/mydeploytool/bin/deploy download -a myService" startCmd="/
myservices/myservice/myservice" workDir="/myservices/myservice/work">
</osType>
...
</Service>
```

5. Click Configure Applications and select the application to modify.

   The Application Profile window displays.
6. Click Import and browse to select the changed application profile, then click Import.

Note:

You can also use the soamreg command to register your new application profile.

7. Click Save to save your changes.

Deploy a service package without a deployment tool

You do not need to use the deployment tool distributed with Symphony to deploy your service packages. You want to skip the deployment step altogether, and still be able to tell Symphony where the service binaries are located on each machine.You may need to do this because you image your compute hosts with the service binaries already on the machines, or service binaries are in a shared location, so deployment is not necessary.

- You do not need to create a service package—you can copy the service binaries and any additional required files to the desired location
- If the service binaries are in a shared location, service binaries and any other additional files required by the service must be accessible to all compute hosts
- If the service binaries are locally installed on compute hosts, the service binaries must be in the same location on all compute hosts
- Symphony grid only. The operating system user account assigned to the consumer for the application must have permissions to execute the service binaries on compute hosts

1. Copy the service binaries to the desired location.
2. In the Platform Management Console, create an application with the Add Application Wizard.
3. Click Configure Applications and select the application to modify.

   The Application Profile window displays.
4. Under Service Definition, Operating System Definition, change the Start Command and Work Directory.
   a) Start Command—Specify the path to your service binary.

      On Windows:

      For example, if the service1 binary is located locally on each machine in `c:\myservice\service1.exe`, specify

      ```
      C:\myservice\service1.exe
      ```

      .

      On Linux/UNIX:

      For example, if the service1 binary is located locally on each machine in `/share/myservice/service1`, specify: **/share/myservice/service1**

   b) Work Directory—Specify the absolute path on the compute host to the directory in which the service creates files.

      On Windows, for example: `C:\myservice\work`.

      On Linux/UNIX, for example: `/share/myservice/work`.
5. Click Save to save your changes and update your application profile.

Automatically run a command when deploying a service package

Suppose your service uses a third-party tool and it needs to be installed on the compute host, or you want to run a script to perform some actions for proper functioning of the service program. You can configure this in a package-specific `deployment.xml` configuration file.

Windows service package

1. Create a file for your service package with the name `deployment.xml`.

   The file must be called `deployment.xml`.

   For example:

```
<Deployment>
     <install>
        <osTypes>
 <osType name="NTX86" startCmd="setup" timeout="600" successCodes="0,1,2"/>
        </osTypes>
    </install>
    <uninstall>
       <osTypes>
          <osType name="NTX86" startCmd="setup -u" timeout="30" successCodes="0"/>
        </osTypes>
   </uninstall>
</Deployment>
```

Note:

To run a Windows .bat script, you need to specify a special syntax.

For example:

```
<osType name="NTX86" startCmd="cmd /c cmd /c
install.bat" timeout="600" successCodes="0,1,2"/>
```

2. Use the install section to configure the command to run after the package is uncompressed on a compute host.
3. For startCmd, specify a path relative to the service package installation directory.

   For example, if your package contained a subdirectory called scripts with the command you want to invoke called `myscript`, specify:

```
startCmd="scripts\myscript"
```

4. Use the uninstall section to configure the command to run if the startCmd specified in the install section fails, or before the package is removed from a compute host.
5. Add `deployment.xml` to your service package with the executables for the commands you specified in `StartCmd`.

Important:

There can only be one deployment.xml file per service package. The file must be at the top level of the service package—it cannot be in a subdirectory.

6. Deploy the service package.
   a) In the Platform Management Console, select Manage Service Packages > Global Actions > Add Package to Repository.

      The Add Package to respository page displays.

  b) Browse to your service package and select it.
  c) Select the application associated with your service package, then Add.

     Your service package should now be displayed in the list.

Note:

You can also use the following commands:

soamdeploy add SampleService -p SampleService.exe.gz -c /SampleApplications/SOASamples

soamdeploy view -c /SampleApplications/SOASamples

Linux/UNIX service package

1. Create a file for your service package with the name `deployment.xml`.

   The file must be called `deployment.xml`.

   For example:

```
<Deployment>
   <install>
     <osTypes>
    <osType name="LINUX86" startCmd="setup" timeout="600" successCodes="0,1,2"/>
      </osTypes>
    </install>
   <uninstall>
     <osTypes>
         <osType name="LINUX86" startCmd="setup -u" timeout="30" successCodes="0"/>
       </osTypes>
   </uninstall>
</Deployment>
```

Note:

All values in `deployment.xml` are case-sensitive when the service is deployed on Linux/UNIX.

2. Use the install section to configure the command to run after the package is uncompressed on a compute host.
3. For startCmd, specify a path relative to the service package installation directory.

   For example, if your package contained a subdirectory called scripts with the command you want to invoke called `myscript`, specify:

   ```
   startCmd="scripts/myscript"
   ```
4. Use the uninstall section to configure the command to run if the startCmd specified in the install section fails, or before the package is removed from a compute host.
5. Add `deployment.xml` to your service package with the executables for the command you specified in `StartCmd`.

Important:

There can only be one deployment.xml file per service package. The file must be at the top level of the service package—it cannot be in a subdirectory.

6. Deploy the service package.
   a) In the Platform Management Console, select Manage Service Packages > Global Actions > Add Package to Repository.

The Add Package to respository page displays.

b) Browse to your service package and select it.
c) Select the application associated with your service package, then Add.

Your service package should now be displayed in the list.

Note:

You can also use the following commands:

```
soamdeploy add SampleService -p SampleService.tar.gz -c /SampleApplications/SOASamples
```

```
soamdeploy view -c /SampleApplications/SOASamples
```

Run multiple services in an application

Goal

In your application, different sessions may need different services to perform computations.

You want to be able to specify that an application can run several services.

Assumptions

For the procedures in this document, it is assumed you want your application to use two different services, ServiceA, and ServiceB. The application is registered under the consumer `/SampleApplications/SOASamples`.

Package and deploy your services

Different services can use separate service packages or the same service package. Ensure each service definition in an application profile has a unique service name. Note that only one of your services can be set as the default service.

This example assumes you will create a separate package for each service used by your application.

For example, to use ServiceA and ServiceB in your application:

- Create `ServiceApkg.gz` and include all related binaries for ServiceA in this package.
- Create `ServiceBpkg.gz` and include all related binaries for ServiceB in this package.

1. Go to the directory that contains your service binaries and compress the service binaries into two files: `ServiceApkg.gz`, and `ServiceBpkg.gz`.
2. Deploy the service packages in the consumer with the `soamdeploy` command. (If you prefer, you may use the Wizard for this task.)

```
soamdeploy add ServiceApkg -p ServiceApkg.gz -c /SampleApplications/SOASamples
soamdeploy add ServiceBpkg -p ServiceBpkg.gz -c /SampleApplications/SOASamples
```

   The service packages are deployed.
3. Check the list of deployed services with the `soamdeploy view` command.

   For example:

   soamdeploy view -c /SampleApplications/SOASamples

   You should be able to see your service packages deployed. Notice that the Application field has a dash (-), indicating that there are no applications associated with the package you deployed.

Associate your application with the service packages

To associate your application with the different service packages, edit your application profile.

Important:

By configuring more than one service in your application, a host blocked because of an error in one of the services is also blocked for all other services of the same application.

1. Open your application profile.

2. In the session type definition, under `SessionTypes`, specify the session type name and add the `serviceName` parameter.

   The `serviceName` parameter can be any name you want. It is used to link the session type definition with the service definition. If `serviceName` is not defined, the session uses the default service.

   For example:

```
...
<SessionTypes>
        <Type name="MysessiontypeA" serviceName="ServiceA" priority="1"
recoverable="false" sessionRetryLimit="3" taskRetryLimit="3"
abortSessionIfTaskFail="false" suspendGracePeriod="100"
taskCleanupPeriod="100"persistSessionHistory="all" persistTaskHistory="all"/>
       <Type name="MysessiontypeB" priority="1" recoverable="false"
serviceName="ServiceB" sessionRetryLimit="3" taskRetryLimit="3"
abortSessionIfTaskFail="false" suspendGracePeriod="100"  taskCleanupPeriod="100"
persistSessionHistory="all" persistTaskHistory="all"/>
</SessionTypes>
```

3. In the service definition, under Service, specify the service name, service package name, and start command for the service. All services specified in SessionTypes should be configured in the Service section.

   a) For Service name, specify the same value that you specified for `serviceName` in the session type.

   b) Specify the packageName parameter and specify the name of the package you deployed.

      You can find out the package name with the command `soamdeploy view`.

   a) Change startCmd to point to your service executable.

      Leave the ${SOAM_DEPLOY_DIR} in your path as this is the deployment directory in the system. If your service is located under a subdirectory, indicate the subdirectory after ${SOAM_DEPLOY_DIR} in the path.

      On Windows:

```
<Service name="ServiceA" description="My Sample Service A"
packageName="ServiceApkg" deploymentTimeout="300">
        <osTypes>
            <osType name="all" startCmd="${SOAM_DEPLOY_DIR}\ServiceA.exe">
            </osType>
        </osTypes>
    </Service>
```

      On Linux:

```
<Service name="ServiceB" description="My Sample Service B"
packageName="ServiceBpkg" deploymentTimeout="300">
        <osTypes>
            <osType name="all" startCmd="${SOAM_DEPLOY_DIR}/ServiceB">
            </osType>
        </osTypes>
    </Service>
```

4. Repeat steps 2-3 for every service that you want to refer to in your application.

5. Add a default attribute for the service that you want to designate as the default so that it is started when the service instance manager starts.

   When specifying multiple services, you must designate one service as the default.

   For example, for ServiceA and ServiceB, your application profile should look similar to the following. Note that in this example, ServiceA is the default service.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Profile xmlns="http://www.platform.com/Symphony/Profile/Application" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" name="">
<Consumer applicationName="SampleApplicationCPP"
consumerId="/SampleApplications/SOASamples" policy="R_Proportion"
taskHighWaterMark="1.0" taskLowWaterMark="1.0" resourceBalanceInterval="5"
sessionSchedulingInterval="500" resourceGroupName="ComputeHosts"/>
...
<SessionTypes>
        <Type name="MysessiontypeA" priority="1" recoverable="false"
serviceName="ServiceA" sessionRetryLimit="3" taskRetryLimit="3"
abortSessionIfTaskFail="false" suspendGracePeriod="100"  taskCleanupPeriod="100"
persistSessionHistory="all" persistTaskHistory="all"/>
        <Type name="MysessiontypeB" priority="1" recoverable="false"
serviceName="ServiceB" sessionRetryLimit="3" taskRetryLimit="3"
abortSessionIfTaskFail="false" suspendGracePeriod="100"  taskCleanupPeriod="100"
persistSessionHistory="all" persistTaskHistory="all"/>
</SessionTypes>

<Service name="ServiceA" description="The Sample Service A"
packageName="ServiceApkg" default="true" deploymentTimeout="300">
        <osTypes>
            <osType name="NTX86" startCmd="${SOAM_DEPLOY_DIR}\ServiceA.exe">
        </osType>
        </osTypes>
    </Service>

<Service name="ServiceB" description="The Sample Service B"
packageName="ServiceBpkg" deploymentTimeout="300">
            <osTypes>
                <osType name="LINUX86"startCmd="${SOAM_DEPLOY_DIR}/ServiceB">
                </osType>
            </osTypes>
        </Service>
</Profile>
```

6. Register the application profile with the `soamreg` command. (If you prefer, you may use the Wizard for this task.)

   For example:

   ```
   soamreg SampleApp.xml
   ```

   The application is registered and enabled.
7. Check that the application is associated with the package with the `soamdeploy view` command.

   ```
   soamdeploy view -c /SampleApplications/SOASamples
   ```

   You should be able to see package names and the associated application names.

Check your client application code and run your client

Check your client application code to ensure:

- The application name specified when connecting to the application is the same as that specified in the application profile
- The session types you specified to create the session must exist in your application profile unless you specified " " for the session types, which means to use the default session type.

  If you want to specify the service name directly to override the service configured in the session type, the service must be configured in your application profile.

1. Check client code to ensure the application name specified in `connect( )` is the same as that specified in the application profile.

   For example, if, in your application profile you have `applicationName="SampleAppCPP"`, your client code must also contain `SampleAppCPP`:

```
...
// set up application specific information to be supplied to the System
ConnectionPtr conPtr = SoamFactory::connect("SampleAppCPP",&securityCB);
...
```

2. Check client code to ensure the session type name specified when creating the session is configured in your application profile.

   For example, create sessions with the session types for ServiceA and ServiceB:

```
...
// Set up session creation attributes for ServiceA
SessionCreationAttributes attributesA;
attributesA.setSessionName("mySessionA");
attributesA.setSessionType("MysessiontypeA");
attributesA.setSessionFlags(SF_RECEIVE_SYNC);
// Set up session creation attributes for ServiceBSessionCreationAttributes
attributesB;attributesB.setSessionName("mySessionB");
attributesB.setSessionType("MysessiontypeB");
attributesB.setSessionFlags(SF_RECEIVE_SYNC);
// Create synchronous sessions
SessionPtr sesPtrA = conPtr->createSession(attributesA);
SessionPtr sesPtrB = conPtr->createSession(attributesB);
...
```

3. Save your client code and recompile.
4. Run your client to submit work to your application.

CHAPTER

13

Application configuration

How configuration affects applications and services

In your client code

Specify	Where	Notes
Application name	When connecting to Symphony	The application name you specify must match the application name indicated in the application profile.
Session type	When creating the session	If you specify a session type when creating the session, the session type must match the session type defined in the application profile.

In the application profile

Specify	Where	Notes
Service package name	In the Service definition	If you are using the Symphony service package deployment tool, you specify the service package name in the service section. The package must be deployed on the repository server. The system uses the operating system to identify the service start command.
Operating system	In the Service definition	You specify the operating system name for the service in the service section. The corresponding service start command will be used for the operating system. The binary is contained in the service package.

Application lifecycle

An application can be used when it is registered with the middleware. It can no longer be used when it is unregistered. There can only be one enabled application per consumer at any one time. You can only submit workload for an application that is enabled.

You register an application by registering the application profile with the Platform Management Console or the `soamreg command`.

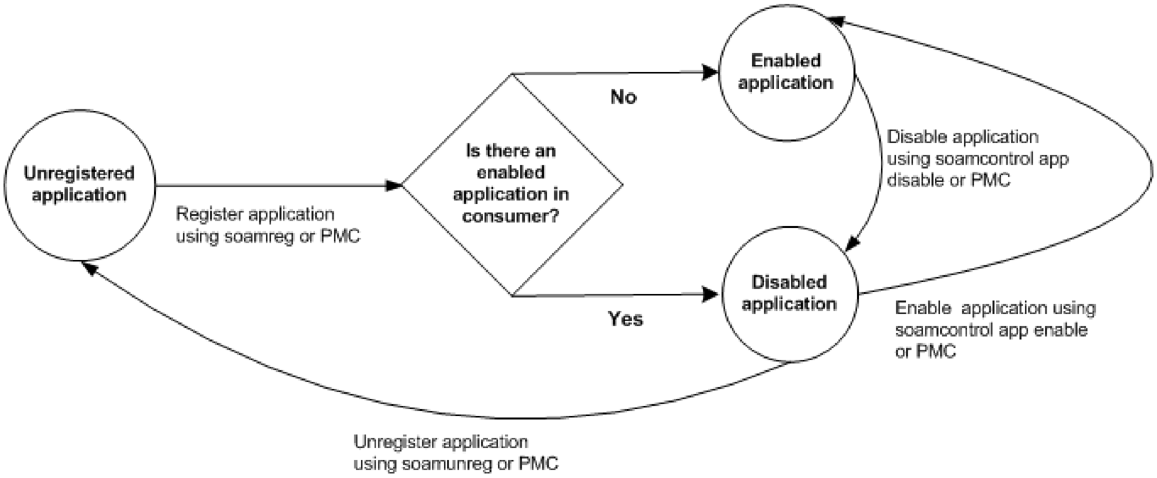
If there is already an enabled application in the same consumer, the application is registered but it is disabled.

If there are no enabled applications in the consumer, the application is registered and enabled.

Once you have registered an application, you can modify its parameters or update service packages through the Platform Management Console or the command-line.

You disable an application with the Platform Management Console or the `soamcontrol app disable` command. When you disable an application, any workload for the application is terminated unless you choose to save workload when disabling it.

You remove an application from the system with the Platform Management Console or the `soamunreg command`. When you remove an application, existing sessions and tasks (running and suspended) are terminated, the application profile is deleted from the system, and all resources allocated to the application are released.


Unregistered application
Register application using soamreg or PMC
Is there an enabled application in consumer?
No
Yes
Enabled application
Disabled application
Disable application using soamcontrol app disable or PMC
Enable application using soamcontrol app enable or PMC
Unregister application using soamunreg or PMC

Change the application profile to only log error historical data

By default, the system displays all data so that you can analyze which tasks have completed when developing client and services.

For performance reasons, you may want to configure your production environment to only enable historical data for tasks in the error state.

Note:

If you are manually editing the application profile outside the Platform Management Console, the parameters that you need to change are in the session type and are called persistTaskHistory, persistSessionHistory. Valid values are all, error, none.

1. In the Platform Management Console, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select the application to modify.

   The Application Profile is displayed.
3. Under Session Type Definition, select a value for Logging History

Setting	Behavior
Log all sessions, error tasks only	Save all session history. Save task history only for those tasks that have completed in error.
Log all sessions, all tasks	Save all session history. Save task history for tasks in all states.
Log all sessions, no tasks	Save all session history. Do not save any task history.
Log no sessions, no tasks	Save no history at all.

4. Click Save to apply your changes.

Specify a different Java location for your application

If your default environment does not point to the correct JRE, add the directory that contains the Java binary to the PATH environment variable in the Service section of the application profile. This adds the directory you specify to the beginning of your existing PATH environment variable.

In cases where you cannot guarantee that the PATH environment variable on a compute host:

- Includes a Java `bin` directory
- Has the path to the Java `bin` directory for the Java version you want to use

You can configure your System PATH to point to the correct Java version, or set the path to the correct version of Java in the application profile.

Note:

If you change the configuration of your system path environment, you must restart Symphony or Symphony DE.

If you set it in your application profile, the directory you specify is added to the beginning of your existing PATH environment variable.

Note:

If you set the Java bin path in your application profile, all compute hosts must have Java installed in the same location. If Java is not installed at that location, the system uses the next Java location in the Path, if any.

1. Open the application profile for your application.

   For example, on Linux:

   `$SOAM_HOME/4.1/samples/Java/SampleApp/SampleAppJava.xml`

   For example, on Windows:

   `%SOAM_HOME%\4.1\samples\Java\SampleApp\SampleAppJava.xml`
2. In the Service, osTypes, osType section, change the PATH environment variable to point to the correct Java version.

```
 <Service description="The Sample Service" name="sampleService">
        <osTypes>
          <osType name="LINUX86" startCmd="${SOAM_DEPLOY_DIR}/
Runcom.platform.symphony.samples.SampleApp.service.MyService.sh" workDir="${SOAM_HOME}/work">
     <env name="PATH">/usr/bin/jdk1.5.0_04_x86/bin</env>
         </osType>
          <osType name="NTX86" startCmd="cmd.exe /c cmd.exe /c ${SOAM_DEPLOY_DIR}/
Runcom.platform.symphony.samples.SampleApp.service.MyService.bat" workDir="${SOAM_HOME}/work"4.1>
     <env name="PATH">c:\java\jdk1.5.0_04_x86\bin</env>
         </osType>
        </osTypes>
</Service>
```

3. Save the file.
4. Update your application profile with the Platform Management Console or the `soamreg` command.

   For example, on Linux:

   soamreg $SOAM_HOME/4.1/samples/Java/SampleApp/SampleAppJava.xml

For example, on Windows:

```
soamreg %SOAM_HOME%\4.1\samples\Java\SampleApp\SampleAppJava.xml
```

Feature: Automatic failure recovery

The automatic failure recovery feature ensures maximum resource availability to run your workload when a system component fails or becomes unavailable due to a power outage, network failure, application deficiency, or other cause.

This feature is not applicable in Symphony DE.

About automatic failure recovery

Purpose of automatic failure recovery

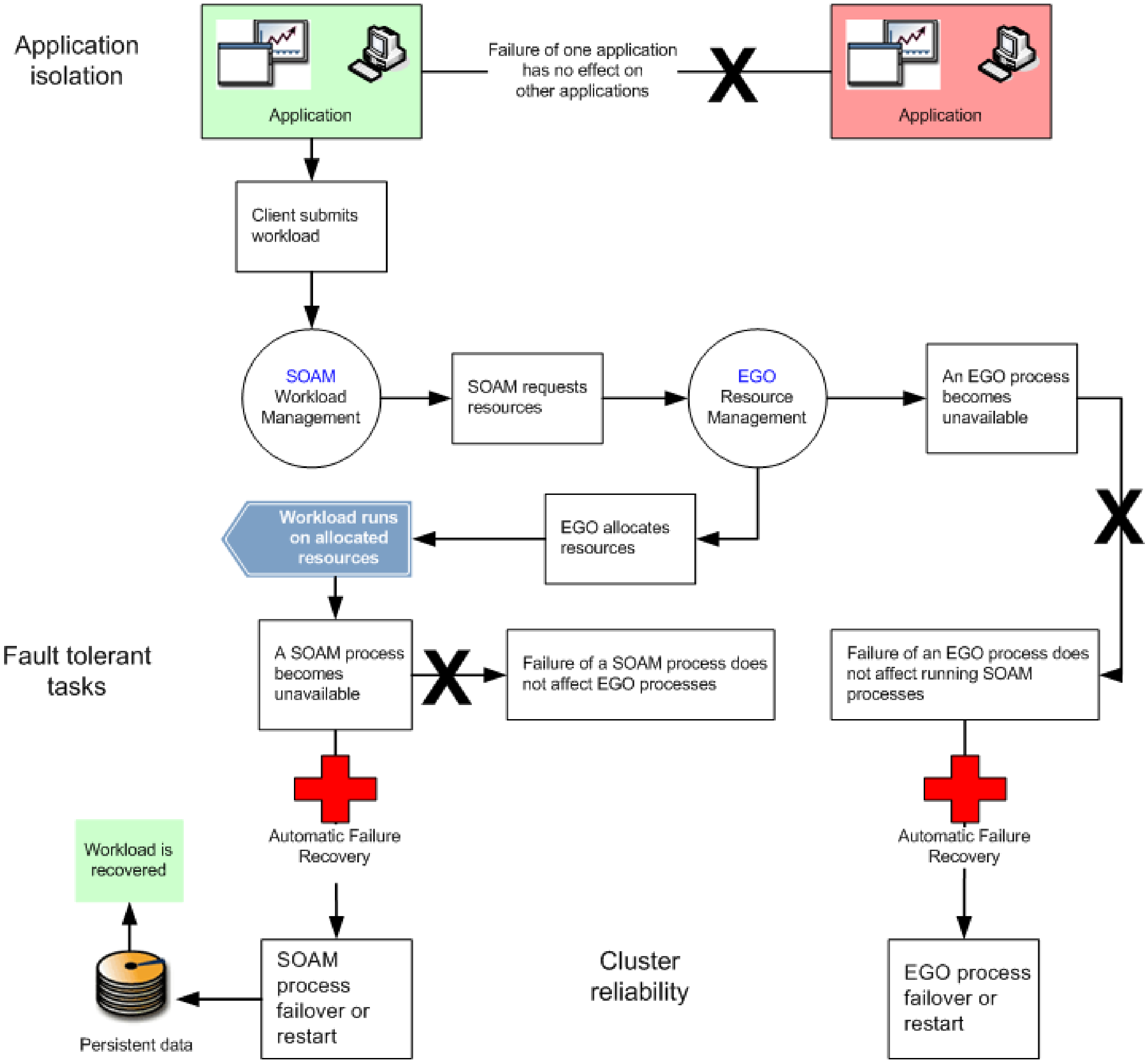
Automatic failure recovery provides a way for the system to automatically restart critical system services and enables you to customize application (service) error handling for each of your applications. Symphony handles a number of failure recovery scenarios.

Benefits of automatic failure recovery

Automatic failure recovery provides a number of benefits, including:

- Application isolation—failure of one application does not affect any other applications, and failure or unavailability of a resource management (EGO) component has no impact on running workload.
- Fault tolerant tasks—with recoverable workload configured, automated failover and data persistence ensures that running workload submitted by an application client continues to run without user intervention when system processes or hosts fail.
- Cluster reliability—master host failover and automatic restart of critical system services ensures high resource availability.

The following illustration shows the benefits of the automatic failure recovery feature once all workload management (SOAM) and resource management (EGO) components have started successfully. In this example, the application profile defines a recoverable session (workload) and the cluster administrator has defined a list of master candidates.


Application isolation
Application
Failure of one application has no effect on other applications
Application
Client submits workload
SOAM Workload Management
SOAM requests resources
EGO Resource Management
An EGO process becomes unavailable
Workload runs on allocated resources
EGO allocates resources
Fault tolerant tasks
A SOAM process becomes unavailable
Failure of a SOAM process does not affect EGO processes
Failure of an EGO process does not affect running SOAM processes
Workload is recovered
Automatic Failure Recovery
Automatic Failure Recovery
Persistent data
SOAM process failover or restart
Cluster reliability
EGO process failover or restart

Scope

Applicability	Details
Operating system	• All host types supported by the Symphony system.
Dependencies	• For master host failover, you must specify one or more master host candidates. • Files required for failover must be on a shared file system. • Cluster administrator and consumer user accounts must have operating system permissions to access directories on the shared file system.
Limitation	• Symphony does not provide automatic failure recovery of the shared file system if the shared file system becomes unavailable.

Configuration to enable automatic failure recovery

Automatic failure recovery is enabled for automatic process restart for critical system services and for restart of Symphony workload management (SOAM) components. Automatic failure recovery for applications is enabled by default in the application profile. You can also enable

- Session manager failover
- Session recovery, which makes workload recoverable
- Master host failover

Configuration to enable session manager failover

Session manager failover is enabled by default when you use a shared file system and do not change the default values for any of the following attributes:

- SOAM > SSM > resourceGroupName
- SOAM > SSM > workDir
- SOAM > DataHistory > path
- SOAM > PagingTasksInput > path
- SOAM > PagingTasksOutput > path
- SOAM > PagingCommonData > path
- SOAM > PagingCommonDataUpdates > path
- SOAM > JournalingTasks > path
- SOAM > JournalingSessions > path
- SOAM > JournalingSessionTagConfig > path

Changing any of these attributes could affect session manager failover. For detailed descriptions of these attributes, see the *Platform Symphony Reference*.

Configuration to enable session recovery

Defining a recoverable session makes workload recoverable after session manager failover or restart.

Section	Attribute name and syntax	Behavior
SessionTypes > Type	recoverable=true | false	• Specifies whether the session can be recovered after session manager failover or restart. If true, Symphony persists the common data and its update (if any) for the session, task data for tasks that have not yet been returned to the client, and data required to reconstruct those objects. If false (default), the system does not persist session and task data, and tasks must be rerun. **Important:** If the file system that is used by the SSM for paging and recovery purposes is not stable, you need to configure flushDataAsap="true" in the application profile. This causes the SSM to write data to disk directly, rather than using system cache, before continuing with the next operation. This guarantees the data is actually stored on the disk and not in system cache after the SSM finishes the disk write operation. The SSM will be able to read the data back in case of recovery. Refer to the application profile reference for details about this parameter. It is strongly recommended to use a stable and reliable file system in your Symphony cluster to avoid losing any data.

Configuration to enable master host failover

The master candidate list defines which hosts are master candidates. By default, the list includes just one host, the master host, and there is no failover. If you configure additional candidates to enable failover, the master host is first in the list. If the master host becomes unavailable, the next host becomes the master, and so on down the list.

For master candidate failover to work properly, the master candidates must share a file system that must always be available.

Important:

The shared file system should not reside on a master host or any of the master candidates. If the shared file system resides on a master host and the master host fails, the next candidate cannot access the necessary files.

If you have configured at least one management host for your cluster in addition to the master host but have not selected any failover candidates, the Platform Management Console dashboard displays a reminder message in red with a link to the page from which you define the master candidate list.

Configuration source	Setting	Behavior
Platform Management Console: **Cluster > Summary >Master Candidates**	• **Add** available hosts to the **Master Candidates** list, or **Remove** hosts from the **Master Candidates** list. • Rearrange the order of master candidates: *host_name* > **Up** | **Down**	• The master candidates are now set in the order you want them to fail over. The cluster automatically restarts when you click **Apply**, making the changes take effect. • All master candidates must be selected from the available management hosts. A compute host cannot be a master candidate. • The default configuration of the EGOManagementServices consumer provides for master candidate failover; do not change the number of slots owned by this consumer.

Alternatively, you can use the command line interface to specify a list of master candidates.

Command	Description
`egoconfig masterlist` *host_name*[,*host_name, …*]	• Specifies the list of master candidates, starting with the master host, and including all of the candidates in the order of failover priority. • *host_name* specifies the name of the master host and each of the master candidates. Do not specify compute hosts in this list. **Caution:** Include all master candidates in the list when you issue this command; `egoconfig masterlist` overwrites the existing list.

Automatic failure recovery behavior

Automatic failure recovery behavior depends on which process fails or becomes unavailable, and on the type of host on which the process runs.

Recovery when individual processes fail or become unavailable

The following description provides details about what happens when a workload management (SOAM), Platform Management Console, reporting, or resource management (EGO) process fails or becomes unavailable independently of other processes.

Important:

Recovery of any workload management (SOAM), Platform Management Console, Reporting, or resource management (EGO) process usually takes less than one or two minutes, and can take as little as one or two seconds, provided that the host remains available.

When this process is in failure recovery…	The effects are…		
	Workload	**Resource allocation**	**Lifecycle or other processes**
Workload management (SOAM) processes			
Service instance (si)	You can define the actions retry or fail for the SessionEnter, SessionUpdate, and Invoke methods.	If blockHost is defined as the actionOnSI for a service instance exit, timeout, exception, or control code, the system terminates the running service instance on this host and does not use this host to start any other service instance for the application. If restart is defined as the actionOnSI, the service instance tries to restart on the original host.	You can define the following actions for service instances based on specific states of the service lifecycle: keepAlive, restartService or blockHost. The session manager will continue to run the service, restart the service on the same host, or—through communications with the Virtual Execution Machine Kernel Daemon (vemkd)—block the host for use by the application associated with the service.
Service instance manager (sim)	The session manager requeues and reruns tasks for the session that was running on the service instance manager; no workload is lost.	If blockHostOnTimeout= "true" in the SOAM > SIM section of the application profile and if, after a service instance manager is started, the service instance manager process cannot contact the session manager within the startUpTimeout, the system does not use this host to start any other service instance managers for the application. If blockHostOnTimeout= "false", the system tries again to start the service instance manager on the original host.	If the service instance manager dies after starting successfully, the associated service instance exits. The session manager then restarts the service instance manager.

When this process is in failure recovery…	The effects are…		
	Workload	**Resource allocation**	**Lifecycle or other processes**
Session manager (ssm)	For recoverable sessions, the session manager persists the information needed to resume the workload without loss of data, and session manager failover or recovery is transparent to the client application. For non-recoverable sessions, the workload is lost and the client must resubmit the workload.	When it restarts, the session manager re-registers with the resource management component (EGO) and obtains a list of resources that were previously allocated to the session manager. The session manager stops and restarts all running service instance managers on those resources.	The service instance managers associated with the failed session manager also die, and requests from the Platform Management Console and command line interface fail. The session director restarts the session manager. On restart, the session manager reads only the task and session control objects, not the input/output messages; the session manager reads those messages as required when dispatching a task. Session manager monitoring information resets; the following statistical values apply to the time period that begins with session manager restart. • Closed sessions since SSM started • Aborted sessions since SSM started • Time of the last session aborted • Done tasks since SSM started • Error tasks since SSM started • Time of the last error task When the session manager is unavailable, clients cannot create new SDK connections. • If the client is already connected and the session manager becomes unavailable, the Symphony APIs retry the connection. • If the client has not yet connected and the session manager is unavailable, the client receives an exception and must wait for the session manager to become available.

When this process is in failure recovery…	**The effects are…**		
	Workload	**Resource allocation**	**Lifecycle or other processes**
Session director (sd)	Session director failure has no impact on running workload; the session manager handles workload execution. For new workload, clients submitting workload wait momentarily for the EGO service controller to restart the session director.	Session director failure has no impact on resource allocation. The session director saves information about the resources it uses and, after restart, uses the same resources.	While the session director is down momentarily, requests from the Platform Management Console and command line interface fail. If you set view preferences for the dashboard to automatically refresh, the request succeeds once the session director has restarted. When the session director is unavailable, clients cannot create new SDK connections. • If the client is already connected and the session director becomes unavailable, the Symphony APIs retry the connection. • If the client has not yet connected and the session director is unavailable, the client receives an exception and must wait for the session director to become available. The EGO service controller usually restarts the session director within a few seconds on the original host or on a new host if the original host has no available resources. The EGO service controller tries up to 10 times to restart the session director before setting the status to ERROR.

When this process is in failure recovery…	The effects are…		
	Workload	**Resource allocation**	**Lifecycle or other processes**
Repository service (rs)	Repository service failure has no effect on running workload. New workload that needs to download a service package must wait until the repository service becomes available.	Repository service failure has no effect on resource allocation.	The EGO service controller restarts the repository service on the original host or on a new host if the original host has no available resources. The EGO service controller tries up to 10 times to restart the repository service before setting the status to ERROR.
Platform Management Console processes			
Web service manager (wsm)	Web service manager failure has no effect on workload.	Web service manager failure has no effect on resource allocation.	The EGO service controller restarts the Web service manager on the original host or on a new host if the original host has no available resources. The EGO service controller tries up to 10 times to restart the Web service manager before setting the status to ERROR. The web service manager monitors the java process of TOMCAT—a key component of the Platform Management Console—and restarts the java process if it goes down.
Reporting processes			

When this process is in failure recovery…	The effects are…		
	Workload	**Resource allocation**	**Lifecycle or other processes**
Platform loader controller (plc)	Loader controller failure has no effect on workload.	Loader controller failure has no effect on resource allocation.	If the loader controller becomes unavailable, the Platform Enterprise Reporting Framework cannot collect sampling data for reporting purposes. The EGO service controller restarts the loader controller on the original host or on a new host if the original host has no available resources.The EGO service controller tries up to 10 times to restart the loader controller before setting the status to ERROR.
Data purger (purger)	Data purger failure has no effect on workload.	Data purger failure has no effect on resource allocation.	If the data purger becomes unavailable, the database could temporarily grow until the data purger recovers and can once again purge the data. The time it takes for the database to run out of space depends on the size of your system. The EGO service controller restarts the data purger on the original host or on a new host if the original host has no available resources.The EGO service controller tries up to 10 times to restart the data purger before setting the status to ERROR.
Resource management (EGO) processes			
Master load information manager (master lim)	Master load information manager failure has no effect on running workload. Clients submitting new workload receive an exception.	The system considers the master host unavailable and a master candidate takes over as master host. During failover to the master candidate, the system does not respond to resource allocation requests.	If no master candidate is available, the cluster is down. The system cannot restart the master load information manager; you can manually restart it, however, using the `egosh ego start all` command.

When this process is in failure recovery…	**The effects are…**		
	Workload	**Resource allocation**	**Lifecycle or other processes**
Virtual Execution Machine Kernel Daemon (vemkd)	Virtual Execution Machine Kernel Daemon failure has no effect on running workload. Clients submitting new workload receive an exception.	During failure recovery, the system does not respond to resource allocation requests.	The master load information manager restarts the Virtual Execution Machine Kernel Daemon.
Process execution monitor (pem)	Process execution monitor failure has no effect on running workload.	Process execution monitor failure has no effect on resource allocation.	The load information manager restarts the process execution monitor on a compute or management host. The master load information manager restarts the process execution monitor on the master host.
EGO service controller (egosc)	EGO service controller failure has no effect on running workload.	EGO service controller failure has no effect on resource allocation.	The Virtual Execution Machine Kernel Daemon restarts the EGO service controller.
Load information manager (lim)	The system considers the host unavailable and terminates workload on the unavailable host. EGO notifies the SOAM component (session director or session manager) that has been allocated to the unavailable host. The session director or session manager stops the service (service instance and service instance manager) on that host and requests another resource.	The system does not allocate any resources on the unavailable host.	The master load information manager restarts the load information manager on the compute or management host.

Recovery when hosts fail

When processes become unavailable in combination because of a hardware failure, you see the following behavior.

Note:

The majority of the time required for failover of compute, management, and master hosts is used to confirm that the host is actually unavailable. This prevents temporary network delays or instability from triggering frequent and unnecessary host switches.

When this host is down…	The effects are…
Compute host	• The following processes become unavailable during failure recovery: • Load information manager • Process execution monitor • Service instance manager • Service instance • When the session manager-service instance manager connection breaks, the session manager requeues the affected tasks. If the session manager does not recognize the broken connection, the resource manager (EGO) notifies the session manager within three minutes that the host is down. • The session manager requests a new resource. • Workload runs on the new compute host.
Management host	• The following processes become unavailable during failure recovery: • Load information manager • Process execution monitor • Session director • Session manager • Repository service • Web service manager • Loader controller • Data purger • In less than three minutes, a new management host takes over and gets configuration information from the shared configuration directory.
Master host	• The following processes become unavailable during failure recovery: • Master load information manager • Virtual Execution Machine Kernel Daemon • Process execution monitor • EGO service controller • Session director • Repository service **Note:** If the session director and repository service can be running on any management host. They will become unavailable during failure recovery only if they are running on the master host. • By default, in less than two minutes, a management host from the master candidates list takes over and gets configuration information from the configuration directory on the shared file system. When the primary master host recovers, it takes over from the master candidate. The load information manager on the primary master becomes the master load information manager, and the Virtual Execution Machine Kernel Daemon and EGO service controller processes on the master candidate host are terminated and restarted on the primary master host. All other EGO services, including SOAM processes remain running on their current host.

Configuration to modify automatic failure recovery

You can modify

- Automatic failure recovery behavior for an application
- Service instance error handling—actions for unexpected exits, timeouts, exceptions, or control codes
- Actions for a timeout between the service instance manager and the session manager

Configuration to modify automatic failure recovery for an application

The following attributes and environment variables can be configured to change the way that automatic failure recovery works once it is enabled for an application.

Configuration source	Setting	Behavior
Application profile: Consumer	flushDataAsap=true | false	• Used for recoverable sessions. Specifies whether or not the session manager caches data before writing to disk. • When set to true, data is not cached, it is immediately written to disk. When set to false (default), data is cached before it is written to disk. **Important:** Setting this parameter to true could substantially degrade performance.
	transientDisconnectionTimeout= *seconds*	• Specifies the number of seconds the session manager waits for the client to reconnect before it aborts the session when the connection between the client and session manager is broken. • Specify an integer equal to or greater than 1. The default value is 10 seconds. • Note that if in a new connection a session that was previously disconnected is opened within the transientDisconnectionTimeout period after the original client exited abnormally, the session is not aborted even if abortSessionIfClientDisconnect is set to true.
	ioRetryDelay=*seconds*	• Specifies the number of seconds to wait before retrying an I/O operation after a previous failure. • Specify an integer equal to or greater than 1. The default value is 1.

Configuration source	Setting	Behavior
Application profile: SOAM > SSM	resReq=**"select(***select_string***)" "select (***select_string*) **order(***order_string***)"**	- Describes the criteria for defining a set of resources to run session managers. Session managers should run on management hosts. When specifying a resource requirement string, you must indicate the select string "select(mg)" so that only management hosts are selected to run session managers. - The default value is "", which specifies any host in the ManagementHosts resource group.
Application profile: SessionTypes > Type	abortSessionIfClientDisconnect=true | false	- Specifies whether the session is aborted if the session manager detects that the connection between the client and the session manager is broken. The default value is true. - Used with the transientDisconnectionTimeout attribute.

Configuration to modify service instance error handling behavior

Section	Method	Attribute name and syntax	Behavior
Service > Control > Method > Timeout	• Register • CreateService • SessionEnter • SessionUpdate	actionOnSI=restartService| blockHost	• Specifies whether to restart the service or block the host on timeout. • The default for Register, CreateService, and SessionEnter, SessionUpdate is blockHost.
	• Invoke • SessionLeave		• The default for Invoke and SessionLeave is restartService.
	• SessionEnter • SessionUpdate • Invoke	actionOnWorkload=retry | fail	• Specifies whether to retry the method (default) up to the number of times configured by the session and task retry limits or abort the session (SessionEnter or SessionUpdate)/fail the task (Invoke). **Note:** The retry count for both SessionEnter and SessionUpdate methods are considered together. For example, if SessionEnter fails once and SessionUpdate fails twice, then the session rerun count is equal to 3.

Section	Method	Attribute name and syntax	Behavior
Service > Control > Method > Exception	• CreateService	actionOnSI=restartService| blockHost	• Specifies whether to restart the service or block the host (default) when the specified exception (failure or fatal exception) occurs.
	• Invoke • SessionEnter • SessionUpdate • SessionLeave	actionOnSI=keepAlive | restartService | blockHost	• Specifies whether to continue running the service (default), restart the service, or block the host when the specified exception (failure or fatal exception) occurs.
	• SessionEnter • SessionUpdate • Invoke	actionOnWorkload=retry | fail	• Specifies whether to retry the method up to the number of times configured by the session and task retry limits or abort the session (SessionEnter or SessionUpdate)/fail the task (Invoke). **Note:** The retry count for both SessionEnter and SessionUpdate methods are considered together. For example, if SessionEnter fails once and SessionUpdate fails twice, then the session rerun count is equal to 3.

Section	Method	Attribute name and syntax	Behavior
Service > Control > Method > Exit	- Register - CreateService - SessionEnter - SessionUpdate	actionOnSI=restartService| blockHost	- Specifies whether to restart the service or block the host on if the service process exits during the execution of the method. - The default for Register, CreateService, and SessionEnter, SessionUpdate, is blockHost.
	- Invoke - SessionLeave		- The default for Invoke and SessionLeave is restartService.
	- SessionEnter - SessionUpdate - Invoke	actionOnWorkload=retry | fail	- Specifies whether to retry the method (default) up to the number of times configured by the session and task retry limits or abort the session (SessionEnter or SessionUpdate)/fail the task (Invoke). **Note:** The retry count for both SessionEnter and SessionUpdate methods are considered together. For example, if SessionEnter fails once and SessionUpdate fails twice, then the session rerun count is equal to 3.

Section	Method	Attribute name and syntax	Behavior
Service > Control > Method > Return	• CreateService • SessionEnter • SessionUpdate • Invoke • SessionLeave	actionOnSI=keepAlive | restartService | blockHost	• Specifies whether to continue running the service (default), restart the service, or block the host when the method returns normally and specified code is returned.
	• SessionEnter • SessionUpdate • Invoke	actionOnWorkload=retry | fail | succeed	• Specifies whether to consider the method task as having reached completion based on a normal return (default), retry the method up to the number of times configured by the session and task retry limits, or abort the session (SessionEnter or SessionUpdate)/fail the task (Invoke). **Note:** The retry count for both SessionEnter and SessionUpdate methods are considered together. For example, if SessionEnter fails once and SessionUpdate fails twice, then the session rerun count is equal to 3.

Section	Method	Attribute name and syntax	Behavior
SessionTypes > Type	• Invoke	taskRetryLimit=*integer*	• Specifies the number of attempts to retry a task before the system fails the task. • The value can be 0 or greater. If you specify a value of 3 (default), the system makes 1 attempt to run the task followed by 3 retries before the task fails.
	• SessionEnter • SessionUpdate	sessionRetryLimit=*integer*	• Specifies the number of times the session can retry binding to the service before the session is aborted. • The value can be 0 or greater. If you specify a value of 3 (default), the system makes 1 initial attempt to run the SessionEnter or SessionUpdate methods followed by 3 retries before the system aborts the session.

Configuration to modify service instance manager-session manager timeout actions

You can change how the system handles a timeout between the service instance manager and the session manager.

Section	Attribute name and syntax	Behavior
SOAM > SIM	blockHostOnTimeout="true" | "false"	• If "true" (default), blocks the host for the application when the service instance manager times out while trying to communicate with the session manager. This means that the services associated with the application run on a different host than the one on which the timeout occurred. If "false", the service tries to restart on the original host. • Used with the startUpTimeout attribute.
	startUpTimeout="*seconds*"	• Number of seconds to wait for the service instance manager to communicate with the session manager. The default is 60 seconds. This parameter works in conjunction with blockHostOnTimeout. • After a service instance manager is started, if the service instance manager cannot contact the session manager within the startUpTimeout and if blockHostOnTimeout="true", the session manager requests a new host from EGO and tries to start a new service instance manager on the new host.

Automatic failure recovery interface

Actions to submit workload

No actions required. For recoverable sessions, session manager failover or recovery is transparent to the application client.

Actions to monitor

You can monitor automatic failure recovery through the Platform Management Console and from the command line. You can also set up SNMP traps to capture system events.

User	Command	Description
• Cluster administrator	From the Platform Management Console Dashboard	• Displays the overall health and drill-down details of the cluster, services, and workload. When a process restarts, the process ID changes.
• Cluster administrator	From the command line: `egosh resource list -m`	• Displays the list of failover candidate hosts in the cluster and identifies which host is currently the master.
• Cluster administrator	From the SNMP trap notifications: • `SYS_FAILOVER_RETRIED`	• The system is trying to restart the session manager or service instance manager.
	• `SYS_SSM_DOWN`	• The session manager goes down abnormally.
	• `SYS_SSM_UP`	• The session manager comes up.
• Cluster administrator	From the SNMP trap notifications: • `SYS_VEMKD_UP`	• To receive this notification, you must first configure EGO_EVENT_PLUGIN=*plugin_name* and EGO_EVENT_MASK=LOG_INFO in `ego.conf`. • Indicates that the master host has failed over to a new master host, or that the cluster has been reconfigured.

You can also check the progress of failure recovery as follows:

Process	User	Command	Description
- Service instance manager - Service instance	- Cluster administrator - Consumer administrator - Consumer user	From the Platform Management Console Dashboard: **Symphony Workload > Monitor Workload >*application_name***	- The presence of a running task indicates that the service instance manager and service instance processes are available. - If tasks are pending but no tasks are running, the service instance manager and service instance processes might be unavailable.
		From the command line: `soamview app` *app_name* -l	- Displays the number of running and pending tasks for all sessions of an application. The presence of a running task indicates that the service instance manager and service instance processes are available. - If tasks are pending but no tasks are running, the service instance manager and service instance processes might be unavailable.
- Session manager	- Cluster administrator - Consumer administrator - Consumer user	From the command line: `soamview app` *app_name*	- The presence of a session manager process ID indicates that the session manager is available.
- Session director - Repository service - Data purger - Loader controller - Web service manager	- Cluster administrator	From the command line: `egosh service list`	- If the process appears in the STARTED state, the process is available.
- Master load information manager - Virtual Execution Machine Kernel Daemon - EGO service controller	- Cluster administrator	From the command line: `egosh service list`	- If the command responds, these processes are available. - If the command does not respond, one of these processes might be unavailable.

Process	User	Command	Description
- Load information manager (non-master) - Process execution monitor	- Cluster administrator	From the command line: `egosh resource list`	- If a host has a status of ok, the load information manager and process execution monitor on that host are available.

Actions to control

Not applicable. Automatic failure recovery does not require user intervention.

Actions to display configuration

User	Command	Behavior
- Cluster administrator - Consumer administrator - Consumer user	From the Platform Management Console Dashboard: **Symphony Workload > Monitor Workload > *application_name* > Application Profile**	- Displays settings for all of the application-level automatic failure recovery configuration.
- Cluster administrator - Consumer administrator - Consumer user	From the command line: `soamview app` *app_name* `-p`	- Displays application profile settings for the selected application.
- Cluster administrator	**Cluster > Summary > Master Candidates**	- Displays a list of master candidates and the order in which failover occurs.
- Cluster administrator	From the command line: `egosh resource list -m`	- Displays the list of failover candidate hosts in the cluster and identifies which host is currently the master.

Feature: Resource reclaim

Resource reclaim—a feature of Platform Symphony's borrowing, lending, and sharing functionality—ensures that consumers can take back their deserved shared or lent resources as needed to meet workload demand.

This is not applicable to Symphony DE.

About resource reclaim

Purpose of resource reclaim

Resource reclaim provides a way for the system to reallocate borrowed or shared resources to a consumer when the consumer has workload demand under any of the following conditions:

- A lending consumer has workload demand that requires the use of all slots owned by the lending consumer
- Share ratios are configured, and an under-allocated consumer (a consumer that is not currently using its *deserved* number of shared slots) has workload demand that requires the use of more slots
- A time based resource plan has time intervals that change the number of owned resources, share ratios and limits, borrowing and lending policies, and borrowing and lending limits for one or more consumers

The system does not always return the same resource that the consumer originally lent. If workload is running on a borrowed resource, the system could reclaim a different physical resource (that meets the resource requirements) from the borrower and allocate that resource to the lending consumer in place of the original resource.

Benefits of resource reclaim

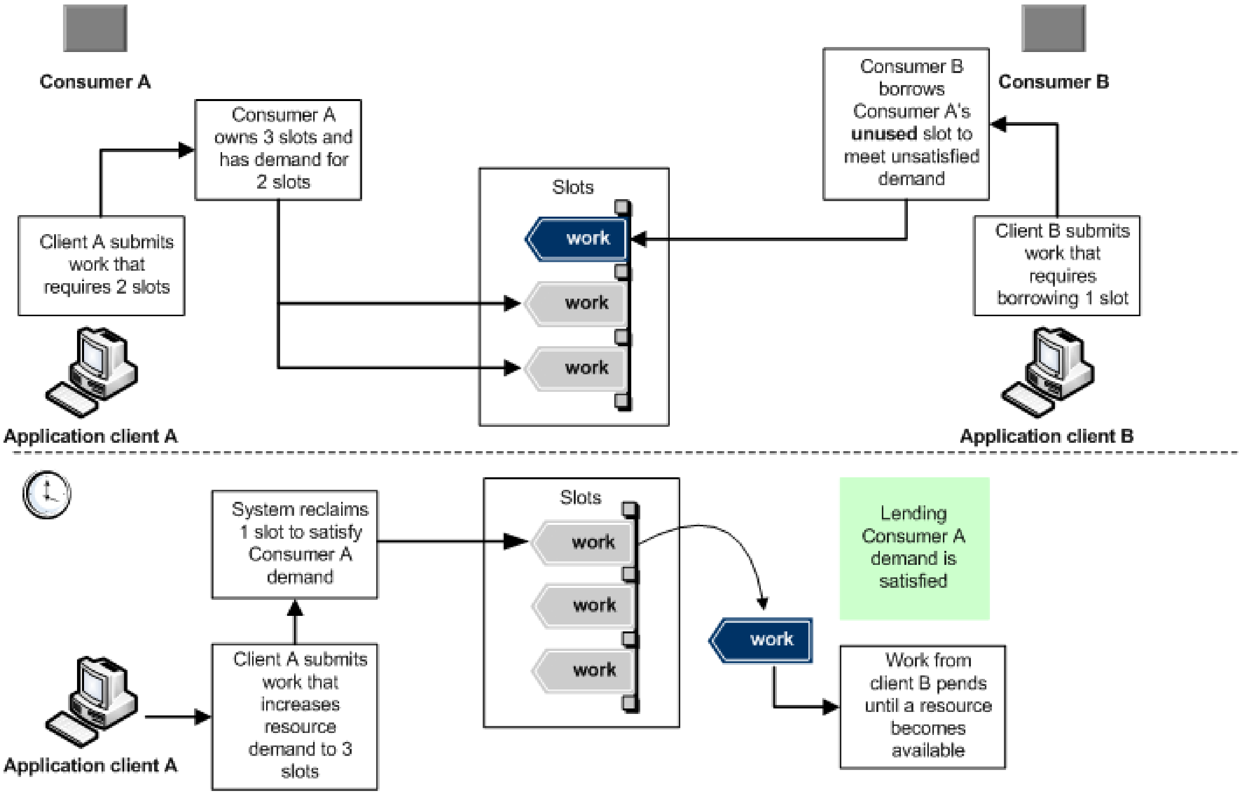
The following illustrations show how the resource reclaim feature works when borrowing, lending, or sharing are enabled.

Important:

Resource reclaim is enabled by default whenever borrowing and lending are enabled. You cannot disable resource reclaim for borrowed or lent resources.

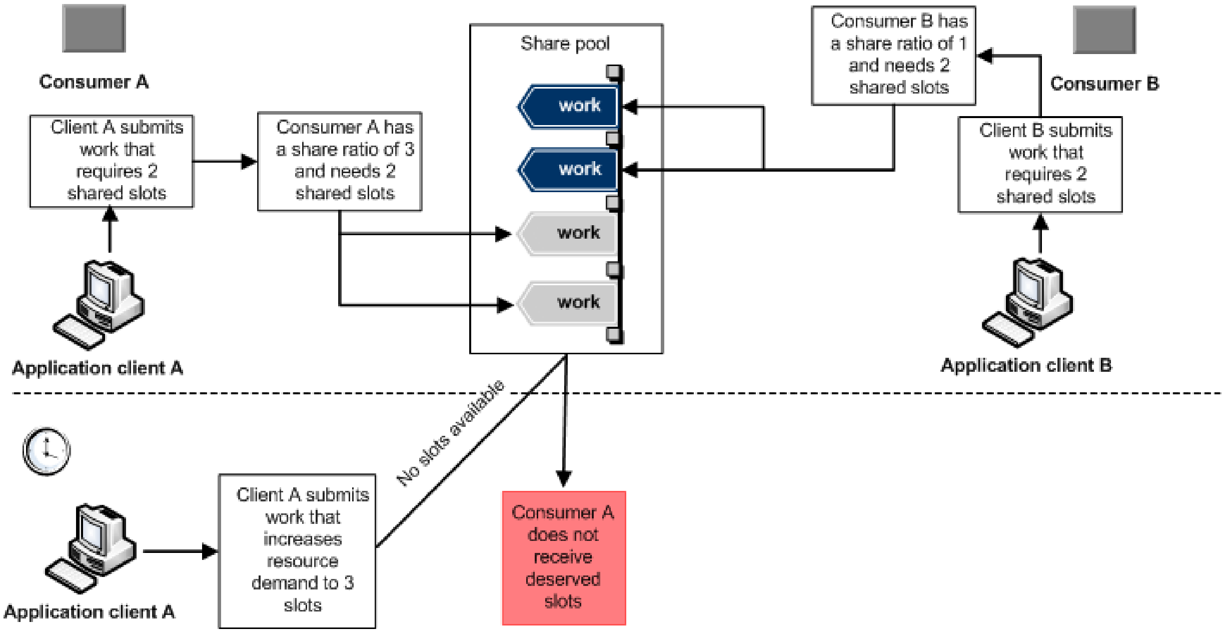
How resource reclaim works for borrowing and lending

You can choose to enable borrowing and lending for owned resources. When you enable borrowing and lending, resource reclaim is always enabled.


Consumer A
Consumer A owns 3 slots and has demand for 2 slots
Client A submits work that requires 2 slots
Application client A
Slots
work
work
work
Consumer B borrows Consumer A's **unused** slot to meet unsatisfied demand
Consumer B
Client B submits work that requires borrowing 1 slot
Application client B
System reclaims 1 slot to satisfy Consumer A demand
Client A submits work that increases resource demand to 3 slots
Application client A
Slots
work
work
work
Lending Consumer A demand is satisfied
work
Work from client B pends until a resource becomes available

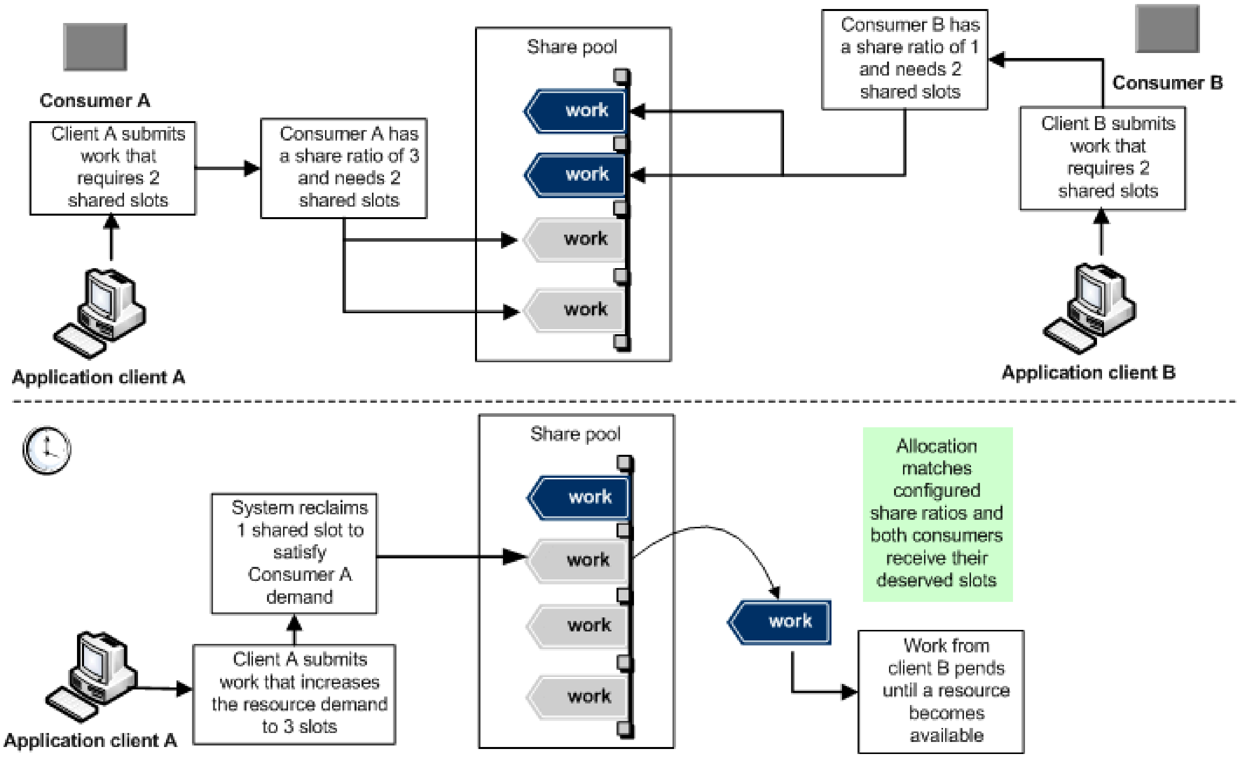
Without resource reclaim for sharing (feature not enabled)

In this example, the share ratio is 3:1. Consumer A deserves 3 times the number of slots as Consumer B.


Consumer A
Client A submits work that requires 2 shared slots
Consumer A has a share ratio of 3 and needs 2 shared slots
Application client A
Share pool
work
work
work
work
Consumer B has a share ratio of 1 and needs 2 shared slots
Consumer B
Client B submits work that requires 2 shared slots
Application client B
No slots available
Client A submits work that increases resource demand to 3 slots
Application client A
Consumer A does not receive deserved slots

With resource reclaim for sharing (feature enabled)

In this example, the share ratio is 3:1. Consumer A deserves 3 times the number of slots as Consumer B.


Consumer A
Client A submits work that requires 2 shared slots
Application client A
Consumer A has a share ratio of 3 and needs 2 shared slots
Share pool
work
work
work
work
Consumer B has a share ratio of 1 and needs 2 shared slots
Consumer B
Client B submits work that requires 2 shared slots
Application client B
Share pool
work
work
work
work
System reclaims 1 shared slot to satisfy Consumer A demand
Client A submits work that increases the resource demand to 3 slots
Application client A
Allocation matches configured share ratios and both consumers receive their deserved slots
work
Work from client B pends until a resource becomes available

Scope

Applicability	Details
Operating system	• All host types supported by the Symphony system
Exclusions	• Does not apply to Symphony DE, which does not have resource lending, borrowing, and reclamation

Configuration to enable resource reclaim

Resource reclaim is enabled whenever you enable lending or borrowing for leaf consumers that own resources. By default, the system

- Immediately sends an interrupt event to the service to notify it of the pending reclaim.
- Allows the service the number of seconds specified in the reclaim grace period to complete processing before terminating the service instance. Tasks that were running on the service instance before it was killed are requeued to their respective sessions. The default grace period is 0 seconds.
- After the reclaim grace period expires, EGO allows 120 seconds leeway time for the return of any reclaimed resources. This is to account for network overhead and other considerations.

- Rebalances resource allocation throughout the cluster when a new time interval begins.
- Reclaims resources under the share model to meet unsatisfied demand within a consumer branch.
- Borrows resources from other consumers before reclaiming lent resources.

Service instance interruption handling

The onServiceInterrupt service handler method provides the most effective way to manage an interruption caused by resource reclaim. Use of this method ensures that the service instance receives immediate notification of a pending interruption.

During a reclaim, the service interrupt indicates how much time the service instance takes to complete current running service method and the service instance to clean up. If the service method and cleanup does not complete within the set time, then Symphony will terminate the instance. If the timeout has not expired, Symphony will initiate cleanup after the current running service method completes.

If a task is running and the Invoke method completes during the applied reclaim grace period, the result of that method is treated as it would be treated under normal circumstances.

If a task is running and the Invoke method does not complete before the applied reclaim grace period expires, the service instance on which the task is running is terminated and the task is requeued.

Another but less effective way to manage an interruption is for the service instance to periodically call the getLastInterruptEvent method for interrupt events. With this method, the service instance polls and will not immediately detect the interrupt. While the service instance is polling, the reclaim grace period is expiring, and the service instance will have less time to return a result or shut down gracefully.

Borrowing and lending with respect to reclaim

Resource reclaim of borrowed resources is always enabled if you configure borrowing and lending at the consumer level. Borrowing and lending can only be configured at the leaf consumer.

Configuration source	Setting	Behavior
Platform Management Console: **Consumers** > **Consumers & Plans** > **Resource Plan > Show Advanced Settings > Expand All**	**Owned Slots**=*integer*	• Specifies a number of slots owned by a leaf consumer. The leaf consumer is guaranteed to receive this number of slots, provided that the consumer has enough demand. If a consumer's owned slots are lent to a borrowing consumer, and the lending consumer has workload demand, the system initiates a reclaim of the owned slots.
	For the lending consumer: • **Lend** checkbox selected • **Details**: • **Lend** checkbox selected for the consumer to lend to	• Enables the consumer to lend resources to the specified consumer(s) • The specified consumer(s) must have borrowing enabled and specify the lending consumer.
	For the borrowing consumer: • **Borrow** checkbox selected • **Details**: • **Borrow** selected for the consumer to borrow from	• Enables the consumer to borrow resources from the specified consumer(s) • The specified consumer(s) must have lending enabled and specify the borrowing consumer.

Share pool and share ratio

Resource reclaim for shared resources is enabled by default once you configure a share pool and share ratios for at least one consumer branch.

Configuration source	Setting	Behavior
Platform Management Console: **Consumers** > **Consumers & Plans** > **Resource Plan > Show Advanced Settings > Expand All**	**Owned Slots**=*integer*	- Specifies a number of slots owned by a leaf consumer. The leaf consumer is guaranteed to receive this number of slots, provided that the consumer has enough demand. If a consumer’s owned slots are lent to a borrowing consumer, and the lending consumer has workload demand, the system initiates a reclaim of the owned slots. - Any unowned slots constitute a “share pool” for allocation to leaf consumers with unsatisfied demand.
	Share Ratio selected and *integer* specified as a value	- Sets the relative share ratios within a share pool. - If you specify 0 for a consumer, that consumer gives up its share of the share pool when a sibling has demand. A consumer with a share ratio of 0 does not receive any resources from the share pool.
Platform Management Console: **Cluster > Summary > Cluster Properties > Specify resource allocation behavior**	**Reclaim shared resources** selected	- When selected (the default setting), the share pool reclaims resources from a consumer that is using more slots than it deserves based on its share ratio to meet the demands of a competing consumer with a higher share ratio.

Resource reclaim behavior

Order of resource reclaim

Consumers reclaim resources in the following order, regardless of a consumer’s history of resource usage:

When the system reclaims …	Then reclaim occurs in the order of…	Example
Borrowed resources	• Resource requirements, determined by the resource group associated with the consumer.	• If the lending consumer needs a Windows slot with a certain amount of available memory, the system looks first for an analogous resource to reclaim.
	• Relative consumer rank, configured in the Resource Plan. Consumer rank is an optional setting. A rank of 1 is the highest rank and larger numbers indicate a lower rank. The system reclaims resources from the lowest ranking consumer first.	• The system first reclaims resources from a consumer with rank 50, and then reclaims resources from a consumer with rank 25.
	• If consumer rank is not configured or if multiple consumers have the same rank, the order of slots reclaimed is random.	• If two consumers both have a rank of 25, each has an equal chance of being reclaimed.
Shared resources	• Relative consumer rank, configured in the Resource Plan. Consumer rank is an optional setting. A rank of 1 is the highest rank and larger numbers indicate a lower rank. The system reclaims resources from the lowest ranking consumer first.	• The system first reclaims resources from a consumer with rank 50, and then reclaims resources from a consumer with rank 25.
	• By default, the system enforces share ratios at the level of the leaf (child) consumers. If your system is configured to enforce share ratios at the parent level, the system reclaims resources from the parent consumer.	• Consumer A is a child consumer of Parent A. Parent A and Parent B are siblings. With share ratio enforced at the parent level, Parent A shares 10 slots with Parent B. Parent B is running workload on 5 slots obtained from Parent A’s share. If Consumer A has unsatisfied demand for 2 slots and all of Parent A’s slots are allocated, the system reclaims 2 slots from Parent B to allocate to Parent A.

Consumer demand

Consumers with workload demand can have lent resources reclaimed for them. When the system reclaims a resource, the system interrupts the borrower’s tasks running on the reclaimed resource. The reclaim grace period allows time for a task running on a borrowed slot to complete before the resource returns to its owner. To avoid being requeued, tasks must exit within the reclaim grace period.

By default, the system reclaims owned resources only after attempting to satisfy demand by borrowing resources from other lending consumers or from the share pool. You can change this behavior so that the system reclaims owned resources before allocating borrowed or shared resources.

Time interval transitions

With a time based resource plan that specifies different values for ownership, lend and borrow limits, share ratios and limits, or total slots in the share pool, a transition from one time interval to the next can trigger resource reclaim. By default, the system enforces ownership and limits when the new time interval takes effect. The following examples illustrate how time interval changes trigger resource reclaim:

When…	The behavior is…	Example
A consumer’s ownership increases for the new time interval, lending and borrowing are not configured, and another consumer is using more than its deserved resources	The system reclaims slots whether or not consumers have unsatisfied demand.	**1.** Consumer A owns 10 slots between 8:00 a.m. and 5:00 p.m. and 25 slots between 5:01 and 11:49 p.m. **2.** At 5:01 p.m., Consumer B is using more than its deserved slots. **3.** At 5:01 p.m., the system reclaims 15 slots to allocate to Consumer A.
A consumer’s ownership decreases for the new time interval, and lending and borrowing are not configured	The system reclaims the number of slots required to conform to the ownership values configured for the new time interval, whether or not other consumers have unsatisfied demand.	**1.** Consumer A owns 10 slots between 8:00 a.m. and 5:00 p.m. and 5 slots between 5:01 and 11:49 p.m. **2.** Consumer B owns 5 slots between 8:00 a.m. and 5:00 p.m. and 10 slots between 5:01 and 11:49 p.m. **3.** At 5:01 p.m., the system reclaims 5 slots from Consumer A, even if Consumer A has unsatisfied demand, and allocates 5 slots to Consumer B.
A consumer’s ownership decreases for the new time interval, borrowing and lending for the consumer are configured, and a lending consumer has slots available	The system reclaims the number of slots required to conform to the ownership values configured for the new time interval, and then the consumer borrows available resources; the resource status changes from owned to borrowed.	**1.** Consumer A owns 10 slots between 8:00 a.m. and 5:00 p.m. and 5 slots between 5:01 and 11:49 p.m. **2.** Consumer B owns 5 slots between 8:00 a.m. and 5:00 p.m. and 10 slots between 5:01 and 11:49 p.m. **3.** At 5:00 p.m., Consumer A has workload running on 10 slots and Consumer B has workload running on 5 slots. **4.** At 5:01 p.m., the system reclaims 5 slots from Consumer A, even if Consumer A has unsatisfied demand, and allocates 5 slots to Consumer B. **5.** Consumer A is configured to borrow from Consumer B, and Consumer B is configured to lend to Consumer A. **6.** Consumer B has no demand for the 5 reclaimed slots. Consumer A borrows 5 slots from Consumer B.
A consumer’s lend limit decreases for the new time interval	The system reclaims the number of slots required to conform to the new lend limit whether or not the consumer has unsatisfied demand.	**1.** Consumer A has a lend limit of 10 slots between 8:00 a.m. and 5:00 p.m. and 5 slots between 5:01 and 11:49 p.m. **2.** Consumer B borrows 10 slots from Consumer A. **3.** At 5:01 p.m., the system reclaims 5 slots from Consumer B and allocates them to Consumer A.

When…	The behavior is…	Example
A consumer's borrow limit decreases for the new time interval	The system reclaims the number of slots required to conform to the new borrow limit, whether or not the lending consumer has unsatisfied demand.	**1.** Consumer A has a borrow limit of 10 slots between 8:00 a.m. and 5:00 p.m. and 5 slots between 5:01 and 11:49 p.m. **2.** Consumer A borrows 10 slots from Consumer B. **3.** At 5:01 p.m., the system reclaims 5 slots from Consumer A to return to Consumer B.
A consumer's share limit decreases	The system reclaims the number of slots required to conform to the new share limit, whether or not a competing consumer has unsatisfied demand.	**1.** Consumer A has a share limit of 10 slots between 8:00 a.m. and 5:00 p.m. and 5 slots between 5:01 and 11:49 p.m. **2.** A share pool is configured for the consumer branch (the parent consumer and its children). **3.** At 5:01 p.m., the system reclaims 5 slots from Consumer A to return to the share pool.
The total number of slots in the share pool decreases	The system reclaims the number of slots needed to maintain share ratios whether or not a competing consumer has unsatisfied demand.	**1.** Consumers A and B each have a share ratio of 1. **2.** The consumer branch owns 10 slots between 8:00 a.m. and 5:00 p.m. and 4 slots between 5:01 and 11:49 p.m. **3.** At 5:00 p.m., Consumer A runs workload on 5 slots, and Consumer B runs workload on 5 slots. **4.** At 5:01 p.m., consumers A and B each return 3 slots to the share pool. **5.** During the new time interval, Consumer A runs workload on 2 slots and Consumer B runs workload on 2 slots.

Configuration to modify resource reclaim behavior

Configuration to modify the reclaim grace period

You can configure a different reclaim grace period behavior for each consumer.

Important:

The borrowing consumer determines the reclaim grace period. When you configure borrowing and lending, ensure that the lending consumer can wait for the maximum reclaim grace period configured for all of its borrowing consumers.

Configuration source	Setting	Behavior
Platform Management Console: **Consumers > Consumers & Plans > *consumer_name* > Consumer Properties > Reclaim behavior**	**Reclaim grace period**= *integer* **Seconds | Minutes | Hours**	• Specifies the wait time before the system interrupts workload running on a borrowed or shared host to reclaim the resource. • To reclaim resources almost immediately, specify 0 seconds. • If you leave the reclaim grace period blank or specify 0, the system uses a default grace period of 0 seconds. • As a best practice, you should specify a realistic value that allows tasks from all of your applications enough execution time and time to clean up to avoid unnecessary interruption. Consider both the typical length of a workload unit run by a borrowing consumer and the urgency of workload demand from the lending consumer.

Configuration to modify system rebalancing behavior

You can configure system rebalancing behavior for each consumer.

Note:

Child consumers do not inherit the value set for the parent consumer.

Configuration source	Setting	Behavior
Platform Management Console: **Consumers > Consumers & Plans > *consumer_name* > Consumer Properties > Reclaim behavior**	**Rebalance when time intervals change** selected	• (Default setting) Enforces ownership, share ratios, and borrowing, lending, and share limits for this consumer when the new time interval takes effect, regardless of consumer demand. • If corresponding lending and borrowing consumers have different rebalancing settings (one is selected and the other is deselected), the consumer with an over-allocation in the new time interval determines which setting the system uses, which determines whether rebalancing occurs.
	Rebalance when time intervals change deselected	• When deselected, the system waits until borrowed resources are returned before enforcing new ownership, share ratios, and borrowing, lending, and share limits for this consumer.

Configuration to modify reclaim behavior for shared resources

You can configure whether the system reclaims shared resources or waits until consumers release shared resources after completing workload tasks, and whether to enforce share ratios at the parent level.

Configuration source	Setting	Behavior
Platform Management Console: **Cluster > Summary > Cluster Properties > Specify resource allocation behavior**	**Reclaim shared resources** selected	• (Default setting) Enables the system to reclaim resources from an over-allocated consumer when a consumer with a higher share ratio has unsatisfied demand.
	Reclaim shared resources deselected	• When deselected, the system does not reclaim shared resources.
`ego.conf`	EGO_PARENT_QUOTA=**Y**	• Enforces share ratios at the parent level, which allows a leaf (child) consumer to have resources reclaimed from another consumer branch, based on the parent consumers' share ratios. By default `EGO_PARENT_QUOTA` is set to **N**. • You must restart EGO on the master and all master candidates after modifying `ego.conf`.

Configuration to modify reclaim behavior for owned resources

By default, consumers borrow resources before their owned resources are reclaimed. You can modify this behavior so that lent resources are reclaimed before borrowing resources from another consumer. This is useful when a consumer's owned resources have specific characteristics required to run the consumer's workload, or when borrowing from a different consumer branch incurs costs based on charge-back policies at your site.

Configuration source	Setting	Behavior
Platform Management Console: **Cluster > Summary > Cluster Properties > Specify resource allocation behavior**	**Reclaim lent resources before borrowing** selected	• Enables reclaim of owned resources before borrowing resources from other consumers.
	Reclaim lent resources before borrowing deselected	• (Default setting) When deselected, consumers with unsatisfied demand borrow resources from other consumers before having their owned resources reclaimed.

Resource reclaim interface

Actions to monitor

You can monitor resource reclaim through the Platform Management Console.

Platform Management Console option	Description
Resources > Monitor Resource Allocation	• Displays a list of consumers along with each consumer's current allocation of owned, shared, and borrowed slots and the consumer's current demand

Actions to control

Once you have configured borrowing, lending, and sharing for your cluster, you cannot directly control or release reclaimed resources. When you modify the resource plan and click Apply, changes take effect immediately and could trigger resource reclaim.

User	Interface	Behavior
• Cluster administrator (EGO)	From the command line: `egosh resource close -reclaim` *resource_name*	• Closes a resource, preventing further allocation. The system reclaims the host before it closes; running workload units terminate as per the configured grace period.
• Application developer	Using the API: onServiceInterrupt	• Notifies the service that the service instance manager has sent an interrupt signal.

Actions to display configuration

User	Command	Behavior
• Cluster administrator • Consumer administrator	From the Platform Management Console: • **Consumers > Consumers & Plans > *consumer_name* > Consumer Properties > Reclaim behavior**	• Displays the settings for **Reclaim grace period** and **Rebalance when time intervals change**
• Cluster administrator • Consumer administrator	From the Platform Management Console: • **Consumers > Consumers & Plans > Resource Plan> Show Advanced Settings > Expand All**	• Displays the ownership, rank, lend, borrow, and share settings for all consumers
• Cluster administrator	From the Platform Management Console: • **Cluster > Summary > Cluster Properties > Specify resource allocation behavior**	• Displays the settings for **Reclaim shared resources** and **Reclaim lent resources before borrowing**

Updating applications

This section describes how to update a Symphony application. There are two approaches to updating applications: static configuration and dynamic configuration. The one you choose depends on the scope of your changes.

Static configuration updates allow you to modify any parameter within the application profile. It offers a wider range of possible changes to application profile parameters than the dynamic configuration update but requires that the application be reregistered after the changes. This results in the termination of running workload associated with the application.

Dynamic configuration updates allow you to update an application without impacting existing clients or workload. Using this method, only changes to the service and session type sections of an application profile can be made. For other changes, the application must be disabled and unregistered, which results in the termination of running workload for that application.

Static configuration update

Use this method of updating an application when you want to change some application parameters and the changes are beyond the scope of a dynamic configuration update.

Updating the application profile does the following:

- Terminates existing sessions and tasks
- In Symphony grid, releases all resources allocated to the application and shuts down session manager. In Symphony DE, there is no resource allocation.
- Re-registers the application profile

Note:

You can also update application parameters through the command-line. Use `soamview app appname -p >`*filename* to send your current profile to a file. Edit it and use `soamreg` to re-register the application.

1. In the Platform Management Console, select Symphony Workload > Configure Applications.
2. Click the application name.

   The application profile displays.
3. From the drop-down list, select Basic Configuration or Advanced Configuration.
4. Modify desired parameters.

   Note:

   If parameters you want to modify are not visible through the Console, export the application profile and modify it with an XML editor, then reimport the updated profile.
5. Click Save to apply your changes and restart the application.

About dynamic application updates

This section provides an overview of the dynamic application update feature.

Symphony’s dynamic application update feature facilitates the administration of service packages and their associations with applications. The Symphony application update features allow you to:

- Deploy an updated service package without stopping the current workload.
- Dynamically update or remove session type and service sections from the application profile without stopping the current workload. Only sessions using the removed sections are affected.
- Dynamically add session type and service sections to the application profile without stopping the current workload.
- Deploy a service package to any level of the consumer tree allowing the service to be shared with all consumers below this level. This enables service packages to be shared among multiple applications linked by the downward path of the consumer hierarchy.

The following table offers guidelines for choosing the right method to perform typical application updates. Within this table, the term workload is defined as existing running tasks and sessions associated with the application that is being updated.

Option	What you want to do	Result
1	**1.** You have an updated service package for an enabled application. **2.** You want current and future workload to use the updated service package. **3.** You want to overwrite the original service package. **Note:** You will not be able to switch back to the original service package. Refer to *Update an existing service package* on page 269	• Workload continues to run with the next scheduled task using the updated service package. • The updated service package has the same name as the original service package. • Once updated, the original service package is no longer available in the repository. • Clients do not need modification.
2	**1.** You have an updated service package for an enabled application. **2.** You want current and future workload to use the updated service package. **3.** You want to be able to easily switch back to the original service package, if necessary. **Note:** You must use a new name for the updated service package so that the repository can store both the original and updated packages. Refer to *Change a service package for an existing service* on page 270	• Workload continues to run with the next scheduled task using the updated service package. • The updated service package has a different name than the original serivce package. • Once it is replaced, the original service package is still available in the repository. • Clients do not need modification.
3	**1.** You have a new service package for an enabled application. **2.** You only want clients that have been notified to use the new service. Refer to *Add a new service and session type* on page 272	• Workload continues to run using the existing service. • Clients that are aware of the new session type can use the new service. • Clients may need modification.

Option	What you want to do	Result
4	**1.** You have a new service package for an enabled application. **2.** You want to use the new service when you create a new session. Refer to *Assign a new service to an existing session type* on page 274	- Workload continues to run using the existing service. - New sessions with the updated session type use the new service. - Clients do not need modification.
5	You no longer need a service or session type and want to remove it from an application. Refer to *Remove a service/session type* on page 276	Any session that uses the removed service or session type is aborted.

Update an existing service package

Perform this task when you want to overwrite an existing service package in the repository. For example, you made a modification to a service binary and would like to replace the existing serivce binary with the new one without disrupting existing clients or workload. Workload continues to run with the next scheduled task using the updated service package.

1. Compile your new service binaries and add them to the service package.
2. From the PMC, select Symphony Workload > Manage Service Packages.
3. Select the relevant consumer (applicable to Symphony grid version only).
4. Select Global Actions > Add package to repository.
5. Click Browse and navigate to your updated service package. Select the package and click Open.
6. Choose whether to use the file name as the package name or enter a new name. In either case, ensure that the package name is the *same* as the original package name.
7. From the Select Application drop-down list, select your application.
8. Click Add.

   A confirmation dialog displays. Click OK.
9. Click Close.

   The page displays your new service package.
10. Verify that the workload is still running (applicable to long-running tasks):
    a) Select Symphony Workload > Monitor Workload.
    b) Click the application name.
    c) Click the session ID.
    d) Verify that the tasks are still running. The update takes effect with the start of the next scheduled task.

Update an existing service package using the CLI

Perform this task when you want to overwrite an existing service package in the repository. For example, you made a modification to a service binary and would like to replace the existing serivce binary with the new one without disrupting existing clients or workload. In this case, when you redeploy a service package that is being used by an enabled application, workload continues to run with the next scheduled task using the updated service package.

1. Compile your new service binaries and add them to the service package.

2. At the command prompt, change your current directory to the directory where the service package is located.

   Note:

   For the following step, ensure that the package name matches the original package name in the repository.

   ---

3. Deploy the service package:

   For example:

   soamdeploy add SampleService -p SampleService.zip -c /SampleApplications/SOASamples

4. Verify that the workload is still running (applicable to long-running tasks):

   For example:

   soamview session SampleApp

Change a service package for an existing service

Perform this task when you want to replace a service package but you also want to be able to easily switch back to the original service package. Workload continues to run with the next scheduled task using the updated service package.

1. Compile your new service binaries and add them to the service package.
2. Associate the new service package with a service:
   a) From the PMC, select Symphony Workload > Configure Applications.
   b) Select the relevant consumer (applicable to Symphony grid version only).
   c) Click the application name.

      The application profile displays.

   d) From the drop-down list, select Dynamic Configuration Update.

      A sub menu displays.

   e) Select Change Service Package/Attributes.
   f) From the Service Package drop-down list, select the new service package.

      If the new service package is not in the list:

      1. Select Add Package to repository.
      2. Click Browse and navigate to the new service package. Select the package and click Open.
      3. Choose whether to use the file name as the package name or enter a new name. In either case, ensure that the package name is *different* than the package name you are replacing.
      4. Click Add.

         An information dialog displays. Click Close.

         The new service package displays in the drop-down list.

   g) Update the Start Command, if necessary.
   h) Click Apply.

      A confirmation dialog displays. Click OK.

      An information dialog displays. Click OK.

   i) Click Close.
3. Verify that the workload is still running (applicable to long-running tasks):
   a) Select Symphony Workload > Monitor Workload.
   b) Click the application name.
   c) Click the session ID.
   d) Verify that the tasks are still running. The update takes effect with the start of the next scheduled task.

Note:

If you need to switch back to the original service package, simply associate the service with the original service package, as described above.

Change a service package using the CLI

Perform this task when you want to replace a service package but you also want to be able to easily switch back to the original service package. Workload continues to run with the next scheduled task using the updated service package.

1. Compile your new service binaries and add them to the service package.
2. At the command prompt, change your current directory to the directory where the service package is located.

Note:

For the following step, ensure that the package name is *different* than the original package name in the repository.

3. Deploy the service package:

   For example:

   soamdeploy add SampleApp_pkg2 -p SampleApp_pkg2.zip -c /SampleApplications/SOASamples
4. Associate the new service package with the service:
   a) Open the application profile with an editor.
   b) Update the service package name. For example:

```
<Profile ...>
    <Service name="ServiceA" description="My Sample Service A"
  packageName="SampleApp_pkg2" deploymentTimeout="300">
    </Service>
...
</Profile>
```

   c) Save the file.
5. Register the application dynamically:

   For example:

   soamreg SampleApp.xml -d
6. Verify that the workload is still running (applicable to long-running tasks):

   For example:

   soamview session SampleApp

Note:

If you need to switch back to the original service package, simply associate the service with the original service package, as described above.

Add a new service and session type

Perform this task when you want to add a new service and session type to your application. For example, you want to restrict the use of a new service only to clients that have been notified about the new session type in your existing application. You want to add this new service and session type to the application without affecting existing clients or workload. New sessions created after this update can use the new service and session type.

This procedure assumes that you have already created a new service package.

1. Add a new service to the application:
    a) From the PMC, select Symphony Workload > Configure Applications.
    b) Select the relevant consumer (applicable to Symphony grid version only).
    c) Click the application name.

       The application profile displays.
    d) From the drop-down list, select Dynamic Configuration Update.

       A sub menu displays.
    e) Select Add Service/Session Type.
    f) In the Service Definition group, click Add.
    g) Enter the service name. Click Add.
    h) Enter a description for the service.
    i) From the Service Package drop-down list, select the new service package.

       If the new service package is not in the list:

        1. Select Add Package to repository.
        2. Click Browse and navigate to the new service package. Select the package and click Open. Click Add.

           An information dialog displays. Click Close.
    j) Update the start command.

       For example:

       ${SOAM_DEPLOY_DIR}/SampleService
2. Add a new session type to the application:
    a) In the Session Type Definition group, click Add.
    b) Enter the new session type. Click Add.
    c) From the Service Definition drop-down list, select the new service that you just added to the application.
    d) Click Apply.

       A confirmation dialog displays. Click OK.

       An information dialog displays. Click OK.
    e) Click Close.
3. Verify that the workload is still running (applicable to long-running tasks):

  a) Select Symphony Workload > Monitor Workload.
  b) Click the application name.
  c) Click the session ID.
  d) Verify that the tasks are still running. (Existing clients and workload are not affected.)

  You can use the new service and session type when you create a new session.

Add a new service and session type using the CLI

Perform this task when you want to add a new service and session type to your application. For example, you want to restrict the use of a new service only to clients that have been notified about the new session type in your existing application. You want to add this new service and session type to the application without affecting existing clients or workload. New sessions created after this update can use the new service and session type.

This procedure assumes that you have already created a new service package.

1. Add the new session type and service to the application:
   a) Open the application profile with an editor.
   b) Add the new session type definition to the application by creating a new Type element.
   c) Set the session type name and service name attributes.

   The service name can be any name you want and is used to link the session type definition with the service definition.

   For example:

```
...
<Profile ...>
    ...
        <SessionTypes>
            <Type name="MysessiontypeA" serviceName="SampleServiceA" priority="1"
            recoverable="false" sessionRetryLimit="3" taskRetryLimit="3"
            abortSessionIfTaskFail="false" suspendGracePeriod="100"
            taskCleanupPeriod="100"persistSessionHistory="all"
            persistTaskHistory="all"/>
        </SessionTypes>
```

   d) Add the new service definition to the application by creating a new Service element.
   e) Set the service name and package name attributes.

   The service name must match the service name that you specified in the Type element.

   f) Change startCmd to point to your service executable.

   Leave the *${SOAM_DEPLOY_DIR}* in your path as this is the deployment directory in the system. If your service is located under a subdirectory, indicate the subdirectory after *${SOAM_DEPLOY_DIR}* in the path.

   On Windows:

```
    <Service name="SampleServiceA" description="My Sample Service A"
  packageName="ServiceApkg" deploymentTimeout="300">
        <osTypes>
            <osType name="all" startCmd="${SOAM_DEPLOY_DIR}\SampleServiceA.exe">
            </osType>
        </osTypes>
    </Service>
...
</Profile>
```

   On Linux:

```
    <Service name="SampleServiceA" description="My Sample Service A"
  packageName="ServiceApkg" deploymentTimeout="300">
        <osTypes>
            <osType name="all" startCmd="${SOAM_DEPLOY_DIR}/SampleServiceA">
            </osType>
        </osTypes>
    </Service>
...
</Profile>
```

   g) Save the application profile.

2. Register the application profile dynamically with the `soamreg` command.

   For example:

   ```
   soamreg SampleApp.xml -d
   ```

   The application is updated, registered, and enabled. Existing clients and workload are not affected by the update. You can use the new service and session type when you create a new session.

3. Verify that the workload is still running (applicable to long-running tasks):

   For example:

   soamview session SampleApp

Assign a new service to an existing session type

Perform this task when you want to assign another service to an existing session type. For example, you have a new service and want to associate it with an existing session type. You want to use the new service starting with the next session.

Updating the session type in the application profile results in the following:

- Open sessions having the updated session type will continue to use the old service until the sessions are closed
- New sessions created after the update will use the new service

This procedure assumes that you have already created a new service package.

1. Add a new service to the application:
   a) From the PMC, select Symphony Workload > Configure Applications.
   b) Click the application name.

      The application profile displays.
   c) From the drop-down list, select Dynamic Configuration Update.

      A sub menu displays.
   d) Select Add Service/Session Type.
   e) In the Service Definition group, click Add.
   f) Enter the new service name. Click Add.
   g) From the Service Package drop-down list, select the new service package.

      If the new service package is not in the list:

      1. Select Add Package to repository.
      2. Click Browse and navigate to the new service package. Select the package and click Open. Click Add.

   h) Click Apply.
2. Associate the new service with an existing session type:
   a) Select Symphony Workload > Configure Applications.
   b) Click the application name.

      The application profile displays.
   c) From the drop-down list, select Dynamic Configuration Update.

      A sub menu displays.
   d) Select Change Service For Session Type.
   e) From the drop-down list in the Session Type Definition group, select the relevant session type.
   f) From the Service Definition drop-down list, select the new service you want to assign to the selected session type.
   g) Click Apply.

      A confirmation dialog displays. Click OK.

      An information dialog displays. Click OK.
   h) Click Close.
3. Verify that the workload is still running (applicable to long-running tasks):
   a) Select Symphony Workload > Monitor Workload.
   b) Click the application name.
   c) Click the session ID.
   d) Verify that the tasks are still running.

      The new service will take effect when you create a new session. Existing clients and workload will not be affected.

Assign a new service to an existing session type using the CLI

Perform this task when you want to assign another service to an existing session type. For example, you have a new service and want to associate it with an existing session type. You want to use the new service starting with the next session.

Updating the session type in the application profile results in the following:

- Open sessions having the updated session type will continue to use the old service until the sessions are closed
- New sessions created after the update will use the new service

This procedure assumes that you have already created a new service package.

1. Add the new service to the application and assign it to an existing session type:
   a) Open the application profile with an editor.
   b) Add the new service definition to the application by creating a new Service element.
   c) Set the service name and package name attributes.
   d) Change startCmd to point to your service executable.

      Leave the *${SOAM_DEPLOY_DIR}* in your path as this is the deployment directory in the system. If your service is located under a subdirectory, indicate the subdirectory after *${SOAM_DEPLOY_DIR}* in the path.

      On Windows:

```
    <Service name="ServiceA" description="My Sample Service A"
  packageName="ServiceApkg" deploymentTimeout="300">
        <osTypes>
            <osType name="all" startCmd="${SOAM_DEPLOY_DIR}\ServiceA.exe">
            </osType>
        </osTypes>
    </Service>
...
</Profile>
```

On Linux:

```
    <Service name="ServiceA" description="My Sample Service A"
  packageName="ServiceApkg" deploymentTimeout="300">
        <osTypes>
            <osType name="all" startCmd="${SOAM_DEPLOY_DIR}/ServiceA">
            </osType>
        </osTypes>
    </Service>
...
</Profile>
```

e) Assign the new service to a session type by setting the serviceName attribute in the Type element to the service name that you specified in the Service element.

For example:

```
...
<Profile ...>
    ...
        <SessionTypes>
            <Type name="MysessiontypeA" serviceName="ServiceA" priority="1"
            recoverable="false" sessionRetryLimit="3" taskRetryLimit="3"
            abortSessionIfTaskFail="false" suspendGracePeriod="100"
            taskCleanupPeriod="100"persistSessionHistory="all"
            persistTaskHistory="all"/>
        </SessionTypes>
```

f) Save the application profile.

2. Register the application profile dynamically with the `soamreg` command.

   For example:

   ```
   soamreg SampleApp.xml -d
   ```

   The application is updated, registered, and enabled. Existing clients and workload are not affected by the update. You can use the new service when you create a new session.

3. Verify that the workload is still running (applicable to long-running tasks):

   For example:

   ```
   soamview session SampleApp
   ```

Remove a service/session type

Perform this task when you want to remove a service or session type and your application is already enabled.

All open sessions having the removed session type or service will be aborted.

1. Select Symphony Workload > Configure Applications.
2. Click the application name.

The application profile displays.

3. From the drop-down list, select Dynamic Configuration Update.

   A sub menu displays.
4. Select Remove Service/Session Type.
5. From th relevant drop-down list, select the service name or session type you want to remove.
6. Click Remove.

   A confirmation dialog displays. Click OK.
7. Click Apply.

   A confirmation dialog displays. Click OK.

   An information dialog displays. Click OK.
8. Click Close.

   The service or session type, as applicable, is removed from the application profile.

Remove a service/session type using the CLI

Perform this task when you want to remove a service or session type and your application is already enabled.

All open sessions having the removed session type or service will be aborted.

1. Remove a service or session type from the application:
   a) Open the application profile with an editor.
   b) Delete the relevant session type or service definition.
   c) Save the application profile.
2. Register the application profile dynamically with the `soamreg` command.

   For example:

   ```
   soamreg SampleApp.xml -d
   ```

   The application is updated, registered, and enabled.

14

Recovery and Performance Tuning

Configure a recoverable session

A recoverable session can be used to preserve your workload under exceptional circumstances such as a power failure or host failure.

Recoverable sessions can incur additional overhead because the workload must be journaled. Specifying your sessions as recoverable may not be appropriate for all types of workload, since it can take less time to rerun all the tasks in the session rather than to recover and resume them. The time it takes to rerun or recover and resume tasks in a session varies with the data size and number of tasks.

Note:

If you are editing the application profile outside the Platform Management Console, set the parameter `recoverable="true"` in the session types section and re-register the application with the soamreg command.

1. In the Platform Management Console,
2. In the Platform Management Console, click Symphony Workload > Configure Applications.

   The Applications page displays.
3. Select the application to modify.

   The Application Profile page displays.
4. In the Session Type Definition section, in Recoverability, select Recoverable.
5. Click Save to apply your changes.

Optimizing session manager performance

Flow control

Flow control prevents session manager from exhausting critical system resources, which may occur under extreme workload.

Flow control does the following:

- Monitors the status of system resources for session manager:
    - Available memory
    - Available virtual address space
- Raises events when a certain threshold has been reached:
    - NORMAL - Operate in default mode for any new input
    - PROACTIVE - Gives early warning to system components that can make memory available when required
    - SEVERE - Starts to scavenge as much memory as possible, current clients work fine
    - CRITICAL -Starts to slow data inflow to the session manager and raise the priority of getting data out of the session manager. Rejects new connections, suspends new sessions from currently attached clients, and pends new tasks in those suspended sessions—current sessions and tasks work fine.
    - HALT - Session manager enters into lockdown mode, no further processing is allowed until an administrative action is performed, or the system enters a safer state.

1. Edit your application profile. In the SOAM SSM section, configure values for memory and virtual address space for each threshold.

   In the example below, when available memory on the session manager host is down to 30% of total memory, the event BEV_PROACTIVE is triggered.

   When available memory is down to 20%, the event BEV_SEVERE is triggered.

   For available virtual address space, when there is only 50% available virtual address space for the session manager process, BEV_PROACTIVE event is triggered, and so on.

   ```
    <boundaryManagerConfig>
                   <boundaries>
                       <boundary elementName="AvailableMemory">
                           <event name="BEV_PROACTIVE" value="30"/>
                           <event name="BEV_SEVERE" value="20"/>
                           <event name="BEV_CRITICAL" value="0"/>
                           <event name="BEV_HALT" value="0"/>
                       </boundary>
                       <boundary elementName="AvailableVirtualAddressSpace">
                           <event name="BEV_PROACTIVE" value="50"/>
                           <event name="BEV_SEVERE" value="40"/>
                           <event name="BEV_CRITICAL" value="25"/>
                           <event name="BEV_HALT" value="15"/>
                       </boundary>
                   </boundaries>
   </boundaryManagerConfig>
   ```
2. Save your application profile.
3. Update your application with the new profile with the `soamreg` command. (If you prefer, you may do these steps using the Platform Management Console to export and import the application profile.)

Memory allocation parameters

If the SSM on Linux remains at a critical memory level when there is enough available memory and not much unfinished workload, the SSM may not be detecting correct memory usage. This can cause the boundary event not to be triggered properly. If this situation occurs, try setting the following environment variables for the SSM in your application profile:

- <env name="MMAP_THRESHOLD">131072</env>
- <env name="MMAP_MAX">65536</env>

Note:

The MMAP_THRESHOLD value should be smaller than the average task input/output size.

CHAPTER

15

Scheduling Configuration

Change the session scheduling interval

The frequency at which the scheduler reassesses resource assignments between sessions is called the scheduling interval.

1. Change the interval in the application profile, by setting the sessionSchedulingInterval attribute in the Consumer section. The attribute is in milliseconds, default 500 milliseconds.

   For example:

```
<Consumer applicationName="MyApplication" consumerId="/consumer" taskHighWaterMark="1.0"
taskLowWaterMark="0.0" preStartApplication="false" numOfPreloadedServices="1"
sessionSchedulingInterval="500" />
```

2. Re-register your application with the `soamreg` command. (If you prefer, do this procedure using the Platform Management Console to export and import the application profile.)

Configure how resources are scheduled by session manager

Platform Symphony supports the following scheduling policies:

- Proportional scheduling
- Minimum services

1. For more details, see the *Symphony Reference*, Application Profile, Consumer section, policy.

Specify criteria for resource selection

When you put your application on the grid in Symphony, you specify a resource requirement string in the application profile to set criteria for selecting a resource or restricting which resources are available to the application.

Note that if you set a resource requirement string, and no hosts match your criteria, no hosts will be available for your application.

Note:

(Not applicable to Symphony DE) To find out the ostype to put into the resource requirement string, use the command egosh resource view with the host name.

1. For example, if you have heterogeneous machines and your service can only run on one type of machine, such as Windows or Linux, set the resource requirement in the `Consumer` section

   For example:

```
 <Consumer applicationName="MyApplication" consumerId="/consumer" taskHighWaterMark="1.0"
taskLowWaterMark="0.0" preStartApplication="false" numOfPreloadedServices="1" resReq="select
(NTX86)"resourceGroupName="ComputeHosts"/>
```

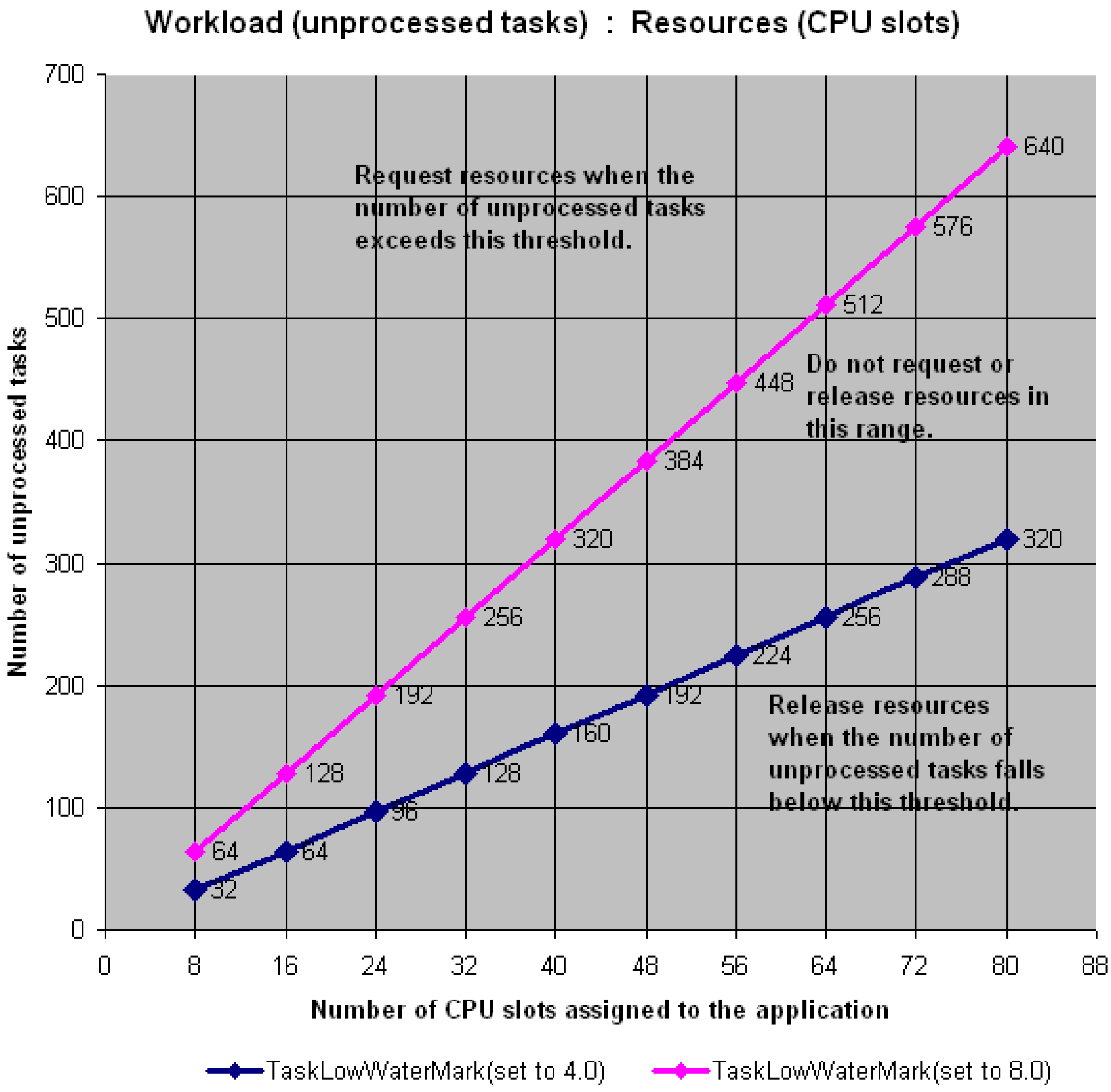
2. Re-register the application with the `soamreg` command. (If you prefer, you can use the Platform Management Console to export and import the application profile.)

Control when applications request or release resources through high- and low-water marks

You can tune your resource requests by specifying a high-water mark and low-water mark in the application profile.

Both high-water mark and low-water mark are expressed as the ratio of the number of unprocessed tasks to the number of CPU slots. Unprocessed tasks include both running and pending tasks.

Together, the high-water mark and low-water mark define a range of satisfactory slot allocation, in which the application does not need to request additional resources or release excess resources.



- High-water mark

  High-water mark allows user to define the threshold for the application as a whole, to request more resources in order to meet its service level requirement. It defines the maximum ratio of unprocessed tasks of open sessions to CPU slots before more resources are requested to meet the service level requirement of the application.

  For example, a high-water mark of 5 means that you want at least one CPU slot for every 5 unprocessed tasks in open sessions.
- Low-water mark allows you to define the threshold for the application as a whole, to return resources that are no longer needed. These returned resources can then be used by other applications.

  The low-water mark is the minimum ratio of unprocessed tasks of open sessions to CPU slots before resources are released and made available for other applications to use.

  The low-water mark should be configured to be less than or equal to the high-water mark.

  For example, a low-water mark of 2 and a high-water mark of 5 means it is satisfactory to have between 2 and 5 tasks for each CPU slot. When there is heavy workload, there can be 5 tasks per CPU slot. When there is light workload, you want to return excess CPU slots until there are at least 2 tasks per CPU slot.

1. High- and low-water marks are configured in the application profile, Consumer section.

   For example:

```
<Consumer applicationName="MyApplication" consumerId="/consumer" taskHighWaterMark="1.0"
taskLowWaterMark="0.0" preStartApplication="false" numOfPreloadedServices="1"/>
```

2. Re-register the application with the `soamreg` command. (If you prefer, you may perform these steps using the Platform Management Console to export and import the application profile.)

Scenario: Maintaining data affinity between a session and service instances

Goal

You have services that cache market data for calculations on compute hosts. Each service loads data into memory and this operation is time-consuming compared to the calculation. Once the data is loaded, it does not change, and it can be used for all calculations that are requested.

Use the minimum services (R_MinimumServices) scheduling policy when you are using common data so that service instances will be reused for tasks in the same session, eliminating the need to reload data for each task.

Change your application profile for data affinity

With this scheduling policy, you define a minimum number of service instances to be allocated to a session, regardless of workload or priority of other sessions, and they continue to serve the session until the session is suspended, killed or closed.

Service instances additional to the minimum service instances are proportionally shared among sessions with pending tasks based on session priority. These service instances are allocated and reallocated to sessions based on priority. Sessions that do not have workload are not allocated additional service instances.

Note:

If you are editing the application profile outside the Platform Management Console, in the Consumer section, add the parameter `policy="R_MinimumServices"`. In the session types section, add the parameters `priority`, and `minServices` and register the application with the `soamreg` command.

1. In the Platform Management Console, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select the application you want to modify.

   The Application profile page displays.
3. Select SSM scheduling policy to expand it, then under Policy Name, select R_Minimum Services.
4. In the Session Type definition, define the Priority for sessions of this type and the Minimum Services (minimum number of CPU slots required for sessions of this type).

   The minimum number of slots remains allocated to the session regardless of workload or priority of other sessions.

   The priority value is used to allocate service instances other than the minimum number of service instances.

   For example, you have 66 service instances and three session types, and you defined the minimum number of instances to be two per session type.

   Two instances are allocated to each session to meet the minimum instance requirement. Then, additional instances are allocated proportionally based on priority.

Session and Session Type	Minimum service instances configured	Priority	Allocated intances	
			Allocated instances (minimum)	Allocated instances (additional)
Session1, SessionA	2	10	2	10
Session2, SessionB	2	20	2	20
Session3, SessionC	2	30	2	30

The sessions receive two service instances each. The remaining 60 service instances are distributed to the sessions proportionally based on priority of the session type.

Session 1 gets 12 service instances in total, 2 gets 22 service instances, and session 3 gets 32 service instances.

5. Click Save to apply your changes.

CHAPTER

16

Client Configuration

Setting your environment on Windows

We assume you are not already using Symphony DE.

Symphony provides batch files and the soamswitch command to facilitate the setup of either your current working environment or your global system environment. The environment setup method you choose depends on how you plan to run your client. If you only want to run your client from your current working command prompt window, use the batch files. If you want to run your client from the Windows desktop or you want to change the system environment permanently, use the soamswitch command.

Change your current working environment

If you want to change your current working environment to connect to a Symphony DE or a Symphony Cluster, run the appropriate batch file:

- **symclientenv.bat**—resets your current command prompt session to connect to the Symphony cluster.
- **symdeenv.bat**—resets your current command prompt session to connect to the DE cluster.

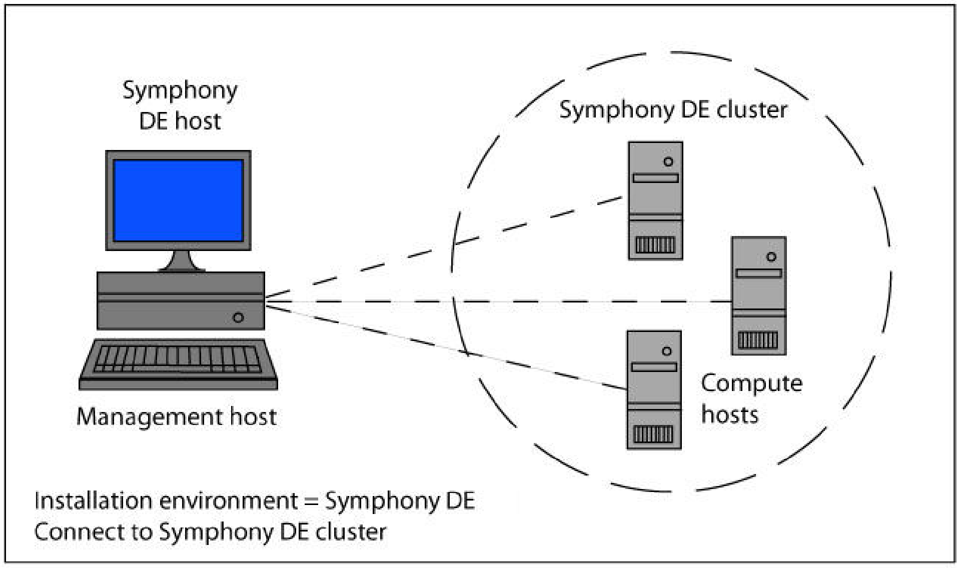
Tip:

When you open the Symphony DE command prompt, the environment is automatically set to DE in the window. You can run `symclientenv.bat` to change the environment to connect to a Symphony cluster from DE. Afterwards, you can run `symdeenv.bat` to change the environment back so that you can connect to a DE cluster.

Change your system environment

Connect to a Symphony DE cluster from the DE environment

This section describes how to set up your Symphony DE system environment so that you can connect to a Symphony DE cluster.


Symphony
DE host
Symphony DE cluster
Compute
hosts
Management host
Installation environment = Symphony DE
Connect to Symphony DE cluster

Note:

You must have local administrator privileges on the host to set the global environment.

Note:

soamswitch does NOT change the environment settings of the command prompt window from which you run soamswitch. The new environment settings only take effect when you open a new Windows command prompt window.

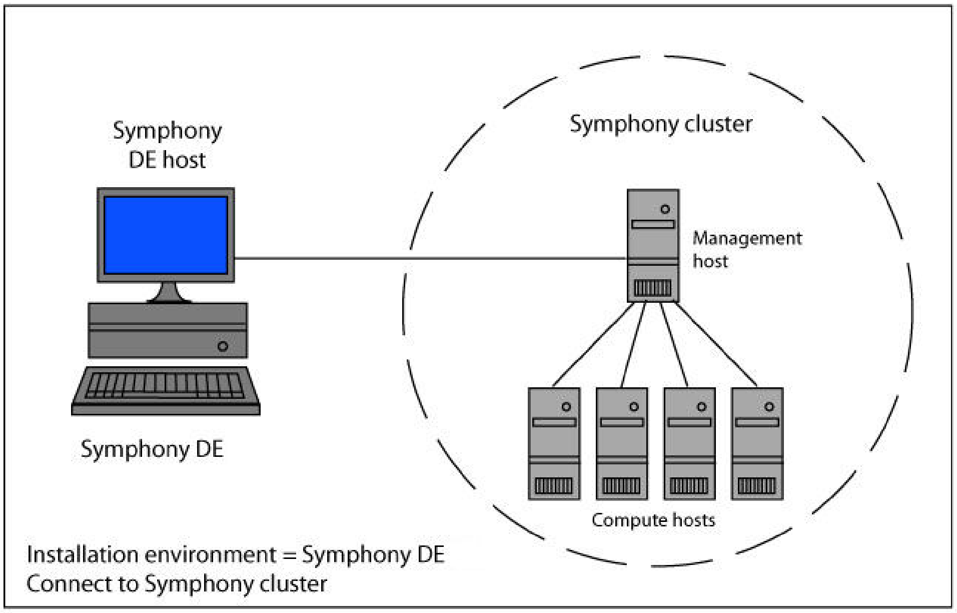
1. At the command prompt, enter:

   soamswitch symde *SymphonyDE_dir*

   For example, **soamswitch symde C:\SymphonyDE\DE40**
2. Open a new Windows command prompt window and run your client from there.
3. Ensure that `vem_resource.conf` is configured properly in the `%SOAM_HOME%\conf` directory.

Connect to a Symphony cluster from the DE environment

This section describes how to set up your Symphony DE system environment so that you can connect to a Symphony cluster.



Note:

You must have local administrator privileges on the host to set the global environment.

Note:

soamswitch does NOT change the environment settings of the command prompt window from which you run soamswitch. The new environment settings will only take effect when you open a new Windows command prompt window.

1. At the command prompt, enter:

   soamswitch sym *SymphonyDE_dir*

   For example, **soamswitch sym C:\SymphonyDE\DE40**
2. Open a new Windows command prompt window and run your client from there.

   Note:

   Do not use the DE command prompt window to run your client since it will reset the environment to Symphony DE.

   Ensure that `ego.conf` is configured properly in%EGO_CONFDIR%, in other words, the `%SOAM_HOME%\conf` directory. Specify the master candidate host list and the EGO `vemkd` daemon port number.

PART

IV

Application Monitoring

CHAPTER

17

Monitoring and Controlling Applications

Log on to the Platform Management Console

This procedure applies to Symphony grid only.

In Symphony DE, a Management Console is installed locally on each host.

The Platform Management Console allows you to monitor, administer, and configure your cluster.

1. If you do not already know the web server URL, run **egosh client view**.
   a) Look for the client name preceded by "GUIURL".

      For example, GUIURL_Host_W (Host_W is the fully qualified host name).

   b) Look for the DESCRIPTION line beneath the client name to find the web server URL, and then copy it.

      For example:
      `http://Host_W:8080/platform`
      .
2. Launch any web browser and enter the address of the web server URL.

   For example:
   http://Host_W:8080/platform .

   The format of the URL is always
   `http://host_name:port_number/platform`
   .
3. Log on to the Platform Management Console for the first time by specifying

   - User Name: **Admin**
   - Password: **Admin**

   For security in a production environment, we strongly recommend that you change the password of the Admin account.

Application monitoring with the dashboard

You can use the Platform Management Console to monitor your applications.

The dashboard is only visible to cluster and consumer administrators. If you are not logged in as a cluster or consumer administrator, you cannot see the dashboard.

Note:

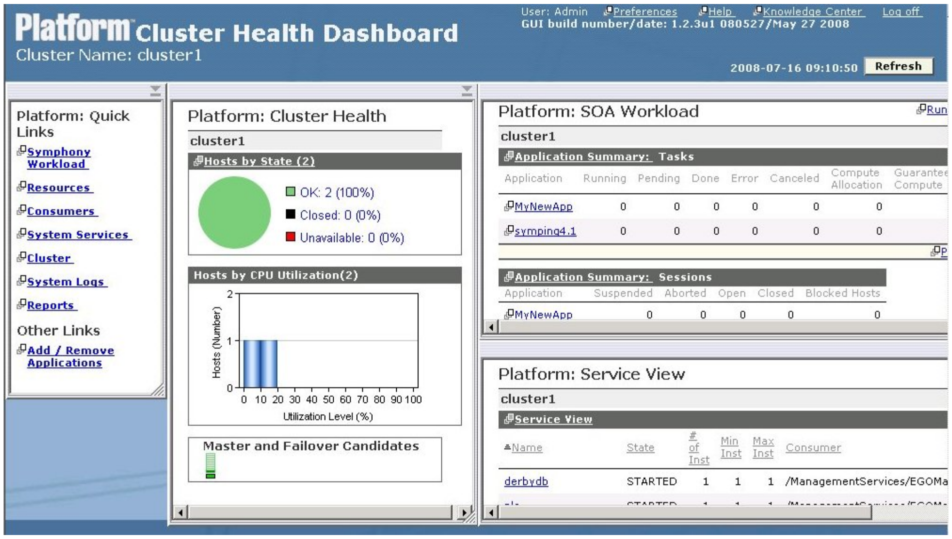
If you installed Symphony DE, the dashboard is not available.

The summary dashboard is your window into the cluster. Use the dashboard to see an overall picture of the health of your cluster and to receive warning for any systemic problems that may be occurring. The dashboard gives you vital, real-time information about the health of your work and your hosts.

Note:

Your dashboard is dynamic and can change depending on what components you have integrated. For information about integration, contact Platform Support.

You can customize your tables on the dashboard and elsewhere by clicking Preferences.



Application	Running	Pending	Done	Error	Canceled	Compute Allocation	Guarantee Compute
MyNewApp	0	0	0	0	0	0	
symping4.1	0	0	0	0	0	0	

Application	Suspended	Aborted	Open	Closed	Blocked Hosts
MyNewApp	0	0	0	0	0

Name	State	# of Inst	Min Inst	Max Inst	Consumer
derbydb	STARTED	1	1	1	/ManagementServices/EGOMa

Symphony command summary

Platform Symphony provides commands for various purposes.

Development environment commands

The following commands are available in Symphony DE.

Command	Description
soamcontrol	Controls applications, sessions, and tasks.
soamdeploy	Deploys, removes, and displays information about service packages for consumers.
soamlog	Dynamically changes the log level for Symphony components.
soammod	Modifies session priority to enable high priority workload to finish faster.
soamreg	Registers an application or updates the application profile of a registered application.
soamunreg	Unregisters an application and deletes the application profile preventing more sessions from being started for this application.
soamstartup	In Symphony DE only, starts Symphony processes on the local host.
soamshutdown	In Symphony DE only, immediately shuts down Symphony processes on the local host.
soamswitch	Windows only. Resets the system environment for the application client: connecting to a DE cluster from the DE installation environment; connecting to a Symphony cluster from the DE installation environment; or connecting to a Symphony cluster using the Symphony installation environment.
soamview	Displays information about applications, sessions, and tasks, and displays the application profile.
symexec	Run executables as Symphony applications.
symping	Sends workload to a cluster to test and verify that Symphony components are working and responsive.

Cluster management and control commands

The following commands are available in Symphony grid.

Command	Description
egosh	Launches the administrative command interface to EGO.
egostartup	(script) Starts all EGO components of a cluster.
egoshutdown	(script) Shuts down a cluster.
pversions	Displays the version information for Platform products installed on a Windows host. This is a Windows command only; this command is not recognized on UNIX systems.

Command	Description
rfa	Transfers files between hosts.
rsdeploy	Deploys and removes middleware packages. You must be a cluster administrator to run this command.
soamlog	Dynamically changes the log level for Symphony components.

Cluster configuration commands

The following commands are available in Symphony grid.

Command	Description
egoconfig	Configures hosts.
egosetrc	Configures automatic startup of EGO on a UNIX host.
egoremoverc	Prevents automatic startup of EGO on a UNIX host.
egosetsudoers	Creates an etc/ego/sudoers file to determine accounts with root privileges on the UNIX host within a cluster.

Workload management commands

The following commands are available in Symphony grid.

Command	Description
soamcontrol	Controls applications, sessions, and tasks.
soamdeploy	Deploys, removes, and displays information about service packages for consumers.
soamlogon	Logs the user on to Platform Symphony for a specific time period.
soamlogoff	Ends the login session with Platform Symphony.
soammod	Modifies session priority to enable high priority workload to finish faster.
soamreg	Registers an application or updates the application profile of a registered application.
soamunreg	Unregisters an application and deletes the application profile preventing more sessions from being started for this application.
soamswitch	Windows only. Resets the system environment for the application client: connecting to a DE cluster from the DE installation environment; connecting to a Symphony cluster from the DE installation environment; or connecting to a Symphony cluster using the Symphony installation environment.
soamview	Displays information about applications, sessions, and tasks, and displays the application profile.
symexec	Run executables as Symphony applications.

Command	Description
symping	Sends workload to a cluster to test and verify that Symphony components are working and responsive.

PART

V

Developer Edition Administration

CHAPTER

18

Managing Symphony DE

Symphony DE quick summary (Windows)

The following section provides a quick summary to start using Symphony DE.

Installation location

The Symphony DE installation directory is `%SOAM_HOME%`. By default, the installation directory is `C:\SymphonyDE\DE40`.

Where to find configuration files

Symphony DE uses the `vem_resource.conf` configuration file to define the type of work that will run on hosts, such as management or compute workload, and the processes that run on hosts such as session manager, session director, GUI service, repository service, and port numbers.

You can find the vem_resource.conf configuration file in:

`%SOAM_HOME%\conf\`

Where to find code samples

Samples are installed with Symphony DE and located in `%SOAM_HOME%\4.1\samples\`.

Where to find documentation and API References

All documentation is installed with Symphony DE. You can access it from the Start menu: Platform Computing > Symphony Developer Edition 4.1.0 > Developer Knowledge Center.

For details on all Symphony APIs, refer to the API References.

Symphony DE quick summary (Linux)

The following section provides a quick summary to start using Symphony DE.

Installation location

The default Symphony DE installation directory is `/opt/symphonyDE/DE40.` When you source the environment, you can access the directory with the `$SOAM_HOME` environment variable.

Where to find configuration files

Symphony DE uses the `vem_resource.conf` configuration file to define the type of work that will run on hosts, such as management or compute workload, and the processes that run on hosts such as session manager, session director, GUI service, repository service, and port numbers.

You can find the `vem_resource.conf` configuration file in:

`$SOAM_HOME/conf/`

Where to find code samples

Samples are installed with Symphony DE and located in `$SOAM_HOME/4.1/samples/`.

Where to find documentation and API References

All documentation is installed with Symphony DE. You can access it from `$SOAM_HOME/docs/symphonyde/4.1/index.html`.

For details on all Symphony APIs, refer to the API references.

Start Symphony DE (Windows)

1. Log on to the host on which you want to start Symphony DE.
2. From a Windows command prompt, run **soamstartup**.

   Note that if you installed Symphony DE without a local administrator account, the command prompt holds the console window. Do not close the window.

   Note:

   If you installed Symphony DE with a local administrator account on all hosts, you can start or stop Symphony DE processes on the local host or on all hosts in your cluster with the Symphony DE Windows tray menus. Right-click on the Symphony DE icon in the Windows tray to display menus. A green color indicates Symphony DE processes are running locally on the host. A blue color indicates Symphony DE processes are not started on the host.

Start Symphony DE (Linux)

1. Got to the directory where you instaledl Symphony DE.

   For example, `/opt/symphonyDE/DE40`

   setenv SOAM_HOME /opt/symphonyDE/DE40
2. Set the environment:
   - For csh, enter

     source /conf/cshrc.soam
   - For bash, enter

     . conf/profile.soam
3. Run **soamstartup &**

Shut down Symphony DE

When you use the `soamshutdown` command, you stop Symphony DE processes on the local host.

1. On the host on which you want to shut down Symphony DE, run **soamshutdown**.

   When you expand your single-host installation to a cluster with multiple hosts, you can use `soamshutdown -all` to shut down Symphony DE on all hosts in your cluster.

Windows

Note:

If you installed Symphony DE with a local administrator account on all hosts, you can start or stop Symphony DE processes on the local host or on all hosts in your cluster with the Symphony DE Windows tray menus. Right-click on the Symphony DE icon in the Windows tray to display menus. A green color indicates Symphony DE processes are running locally on the host. A blue color indicates Symphony DE processes are not started on the host.

Expand a single host installation to a cluster

When you first install Symphony DE, you install on a single host. Before you can add more hosts to create a cluster, you need to convert your single host to a management host. This topic guides you through the process of converting your host to a management host, and then adding compute hosts to create the cluster.

Convert your single host to a management host

A management host is a host that runs processes to schedule and manage workload: the session director process, repository service, GUI service, and the session manager process run on the management host.

1. Log on to the single host on which you installed Symphony DE.
2. If Symphony DE processes are running on the host, shut down Symphony DE:

   soamshutdown
3. Edit the `vem_resource.conf` configuration file (located in the `conf` directory under `SOAM_HOME`).
4. Look for the following lines and replace `localhost` with the actual name of your host. For example:

   ```
   ...
   SD_SDK:15051:myhost:sd
   SD_ADMIN:15050:myhost:sd
   ...
   RS_DEPLOY:15052:myhost:rs
   ...
   WEBGUI:18080:myhost:startguiservice
   ```
5. Look for the AGENT line and replace `localhost` with the actual name of your host. For example, on Windows:

   ```
   AGENT:8000:myhost:5:0:NTX86:1
   ```

   Note that the number of slots for SIMs is set to 0 to prevent application workload from running on the management host. In this example, the management host runs the session director process, the repository service, the GUI service, and up to 5 session manager processes.
6. Save the file.

Install Symphony DE on a compute host

A compute host is a host that runs services and performs computations.

If this is the first compute host you are adding to your cluster, you must first convert your single host to a management host.

1. Install Symphony DE on the host that you want to add to your cluster.

Configure management host to recognize compute hosts

1. Log on to the management host.
2. If Symphony DE is running on the management host, shut down the processes on the management host:

   soamshutdown

3. Edit `vem_resource.conf` (located in the `conf` directory under `SOAM_HOME`) and add one AGENT line per compute host:

```
...
AGENT:8000:myhost:5:0:NTX86:1
AGENT:8000:mysecondhost:0:5:NTX86:1
AGENT:8000:mythirdhost:0:5:NTX86:1
```

4. For each new AGENT line, ensure the maximum session managers and maximum service instance manager values are set correctly:
   a) In the first AGENT line (the management host), set the values to 5:0, indicating that up to five session managers and no application workload can run on the management host.
   b) For each compute host, set the values to 0:5, indicating that no session managers, and up to five service instance managers per application can run on the compute host.
5. Save `vem_resource.conf`.
6. Start up Symphony DE processes on the management host.

   On Windows:

   soamstartup

   On Linux:

   soamstartup &

Synchronize configuration files

1. All hosts in the cluster should use the same configuration. To synchronize the files, you may copy the `vem_resource.conf` file from the management host to the shared location on each compute host, overriding the existing configuration on each compute host.

   If you make any changes to `vem_resource.conf` in future, such as changing ports, you must synchronize the configuration across all compute hosts again.
2. Start up Symphony DE processes on each new compute host.

   soamstartup

Note:

On a Windows host, if you installed Symphony DE with an account that is not a local administrator, soamstartup holds the window. Closing the window shuts down Symphony DE.

Note:

On a Windows host, if you installed Symphony DE with a local administrator account on all hosts, you can start or stop Symphony DE processes on the local host or on all hosts in your cluster menus on the Symphony DE Windows tray. Right-click on the Symphony DE icon in the Windows tray to display menus. A green color indicates Symphony DE processes are running locally on the host. A blue color indicates Symphony DE processes are not started on the host.

Remove a Windows compute host

You want to remove a compute host from the Symphony DE cluster. This topic guides you through the process of uninstalling the software and removing the host from the cluster.

Uninstall Symphony DE

1. Shut down Symphony DE processes on the local host:

   soamshutdown

   Note that `soamshutdown all` shuts down Symphony DE processes on all hosts in your cluster.
2. To uninstall Symphony Developer Edition , use the Windows Add/Remove Programs application.
3. Manually remove any directories in `%SOAM_HOME%`.

   For example, manually remove `C:\SymphonyDE\DE40`, the Symphony DE installation directory.

   Uninstalling does not remove any added or changed files.

Remove the compute host from the cluster

To remove a compute host from the cluster, you need to remove the configuration entry for the host from the `vem_resource.conf` file of the management host.

1. Log on to the management host.
2. Shut down Symphony DE processes on the management host:

   soamshutdown

   Note that `soamshutdown` shuts Symphony DE processes on all hosts in the cluster.
3. Edit `%SOAM_HOME%\conf\vem_resource.conf` and remove the AGENT line that indicates the compute host.
4. Save the file.
5. Start up Symphony DE processes on the management host:

   soamstartup
6. Log on to each compute host and start up Symphony DE:

   soamstartup

Remove a Linux compute host

You want to remove a compute host from the Symphony DE cluster. This topic guides you through the process of uninstalling the software and removing the host from the cluster.

Uninstall Symphony DE

There are two ways to uninstall Symphony DE, depending on which installation files and procedure you installed with.

If you installed using .tar.gz package

1. Shut down Symphony DE processes:

   soamshutdown
2. Delete the Symphony DE installation directory and all subdirectories.

   rm -r $SOAM_HOME

If you installed using RPM

1. Shut down Symphony DE processes on the local host:

   soamshutdown
2. Uninstall:

   - For example, for Red Hat Enterprise Linux 4 or SuSE Linux Enterprise Server 9, enter (on one line)

     rpm -e symphonyDE-linux2.6-glibc2.3-x86-4.1.0.*`build_number`*
   - For example, for Red Hat Enterprise Linux 3, enter

     rpm -e symphonyDE-linux2.4-glibc2.3-x86-4.1.0.*`build_number`*
   - For Red Hat Linux Advanced Server 2.1 or SuSE Linux Enterprise Server 8, enter

     rpm -e symphonyDE-linux2.4-glibc2.2-x86-4.1.0.*`build_number`*

   Note:

   If you used `--dbpath` during installation, to uninstall specify --dbpath. Also, if you use RPM Version 4.1.1, and have installed other software packages using the default dbpath, to uninstall specify `--nodeps`.
3. Manually remove any directories left over in $SOAM_HOME.

   For example, manually delete `/opt/symphonyDE/DE41`, the Symphony DE installation directory.

   Uninstalling does not remove any added or changed files.

Remove the compute host

To remove a compute host from the cluster, you need to remove the configuration entry for the host.

1. Log on to the management host.
2. Shut down Symphony DE processes on the management host:

   soamshutdown

3. Edit `$SOAM_HOME/conf/vem_resource.conf` and remove the AGENT line that indicates your host.
4. Save the file.
5. Start up Symphony DE processes on the management host:

   soamstartup &
6. Log on to each compute host and start up Symphony DE:

   soamstartup &

Index