Построенные над SObjectizer библиотеки [версия 0.4]

Евгений Охотников
Intervale

20 октября 2008 г.

Оглавление

Часть I

Введение

Глава 1

Расширение через библиотеки

SObjectizer [SO4] является относительно небольшой библиотекой, которая позволяет объявить отдельные сущности в приложении агентами и обеспечивает взаимодействие агентов посредством обмена сообщений и приоритетной обработки агентами поступающих к ним сообщений. Это база, на которой могут строится различные типы приложений от небольших программ для работы со SmartCard-ридерами до объемных распределенных вычислений и больших телекоммуникационных систем. Но, на практике, каждый тип приложений накладывает ряд собственных требований и довольно часто возможности SObjectizer оказываются слишком низкоуровневыми. Например, взаимодействие распределенных приложений через сообщения глобальных агентов может быть достаточным только для относительно простых приложений, вроде распределенных вычислений. В более сложных и больших приложениях требуются более удобные средства обмена сообщениями.

Обычно для того, чтобы SObjectizer мог успешно применяться в конкретной предметной области требуется добавить в SObjectizer некоторую специфическую функциональность. Это можно делать либо расширяя сам SObjectizer, либо создавая специализированные библиотеки (фреймворки). Оба подхода имеют свои достоинства и недостатки, но для развития SObjectizer более предпочтительным представляется разработка дополнительных библиотек без модификации ядра SObjectizer. Такой подход позволяет оставить в SObjectizer набор самых базовых понятий, что упрощает как развитие и совершенствование самого SObjectizer, так и существенно облегчает освоение SObjectizer разработчиками, поскольку для овладения SObjectizer в этом случае требуется изучение небольшого числа базовых принципов, понятий и приемов программирования. Библиотеки же могут изучаться в процессе их использования. Причем, только те библиотеки, которые нужны программисту для решения конкретной задачи.

Образно говоря, расширение SObjectizer определяется двумя основными принципами, которые уже давно известны:

- не нужно включать в ядро то, что может быть реализовано в библиотеке. Этот принцип широко используется при развитии языка C++ [CPPD];
- библиотека должна делать всего одну вещь, но делать это хорошо. Данный принцип является основой так называемого Unix way [ATUP].

Оба принципа успешно зарекомендовали себя за более чем двадцатилетнюю историю языка C++ и более чем тридцатилетнюю историю операционной системы Unix. Поэтому они применяются и для развития SObjectizer.

Глава 2

Краткий обзор существующих библиотек

2.1 SObjectizer System Configurator

Библиотека *so_sysconf* предназначена для того, чтобы позволять собирать SObjectizer-приложения из отдельных частей как из конструктора. Она строится на возможности динамической загрузки DLL с кодом новых агентов.

Базовым понятием в *so_sysconf* является понятие DLL. Каждая DLL может содержать т.н. обработчики кооперaций (*coop_handler*) и фабрики кооперaций (*coop_factory*). Разница между *coop_handler* и *coop_factory* в том, что *coop_handler* отвечает за регистрацию и дерегистрацию одной кооперации агентов с заранее заданным именем. *coop_factory* же отвечает за регистрацию и дерегистрацию кооперaций, имена которых становятся известными только в во время работы приложения (т.е. являются обычными фабриками объектов).

Библиотека *so_sysconf* состоит из нескольких кооперaций и вспомогательных инструментов. Для использования *so_sysconf* в SObjectizer-приложении необходимо зарегистрировать основную кооперацию из *so_sysconf*. После этого команды на загрузку/выгрузку DLL и регистрацию/дерегистрацию кооперaций можно отдавать подсистеме *so_sysconf* двумя способами:

1. Через сообщения специального глобального агента. Например, отсылка сообщения `msg_load_dll` предписывает подсистеме *so_sysconf* выполнить загрузку указанной DLL, а сообщение `msg_reg_coop` — выполнить регистрацию кооперации посредством указанного *coop_handler*-а.

2. Через конфигурационный файл, в котором описываются загружаемые библиотеки и регистрируемые кооперации.

Обычно на практике применяется второй способ: для приложения создается конфигурационный файл, в котором фиксируются все составные части приложения (DLL и кооперации). Стартовая часть приложения представляет из себя небольшой загрузчик, задачей которого является старт SObjectizer и подсистемы *so_sysconf*. После чего загрузчик просто передает *so_sysconf* имя конфигурационного файла, а *so_sysconf* выполняет загрузку всех остальных частей приложения. Поскольку это типовой сценарий работы многих SObjectizer-приложений, в *so_sysconf* входят несколько готовых подобных загрузчиков.

2.2 Generic Monitoring Tool

Библиотека *gemont* (Generic Monitoring Tool) предназначена для организации средств мониторинга состояния SObjectizer-приложения. Если SObjectizer применяется для создания приложений без интерактивного пользовательского интерфейса (в виде черного ящика), то возникает вопрос: «А как узнать, что происходит внутри приложения?». Например, если SObjectizer-приложение представляет из себя некую *server-side* систему обработки транзакций, то было бы полезно знать, в каком состоянии находятся основные агенты внутри приложения, сколько транзакций обрабатываются в данный момент, сколько транзакций находятся в очередях и т.д.

Если приложение сохраняет свою информацию в реляционной СУБД, то подобные средства мониторинга можно строить не затрагивая само SObjectizer-приложение. Например, можно разработать Web-приложение (на Java, ASP, PHP или Ruby), которое будет периодически извлекать информацию из СУБД и показывать текущие значения в браузере клиента. Однако, такой подход имеет два недостатка:

- SObjectizer-приложение может не работать с РСУБД и хранить всю текущую информацию либо в специализированных хранилищах (таких как Berkeley DB [BDB] или SQLite [SQLITE]), либо в оперативной памяти, либо вообще не хранить информацию являясь *stateless* приложением;
- такое приложение, периодически извлекающее информацию из РСУБД, не будет показывать информацию в реальном времени.

Идея *gemont* заключается в том, чтобы в SObjectizer-приложение включать специальные *gemont*-источники данных (небольшие объекты, которые хранят в себе значения элементарных типов, таких как *int* или *string*). Весь фокус в том, что *gemont* создает специального глобального агента, который знает про все источники данных и умеет рассылать их текущие значения по запросу, а так же отсылает их новые значения при обновлении.

Когда SObjectizer-приложение нуждается в мониторинге, при его разработке используется *gemont* для представления наиболее важной информации в виде источников данных *gemont*. Например, если требуется наблюдать изменение счетчика транзакций, то этот счетчик реализуется в виде специального *gemont* объекта. Если нужно знать о текущем состоянии какого-то агента, то с данным агентом связывается другой *gemont* объект.

Для мониторинга SObjectizer-приложения, в котором есть *gemont*-источники данных используются специальные инструменты, которые подключаются к SObjectizer-приложению обычным образом (через SOP протокол).

2.3 SObjectizer Logger

Библиотека *so_log* предназначена для предоставления SObjectizer-приложениям возможности фиксирования следа своей работы в т.н. log-файлах. К сожалению, в C++ нет стандартной библиотеки для логирования, поэтому каждый более-менее серьезный фреймворк предоставляет собственный вариант, а так же существует некоторое количество отдельных библиотек, решающих данную задачу. Библиотека *so_log* была одной из первых вспомогательных библиотек для SObjectizer и основной ее целью была простота, компактность, отсутствие лишних зависимостей и использование специфических возможностей SObjectizer. Вероятно, если бы ACE применялась в SObjectizer с самого начала, то надобности в *so_log* не возникло. Тем не менее, *so_log* существует и может использоваться, если проекту вполне достаточно ее функциональности. Так же привлекательной чертой *so_log* является использование для формирования логируемых сообщений стандартных потоков C++ (*iostreams*), что позволяет легко включать в сообщения содержимое объектов, для которых определен оператор сдвига в *std::ostream*.

В отличии от других средств логирования, построенных под впечатлением от Log4J [L4J], в *so_log* используются два типа сообщений: логические и сообщения об ошибках. Сообщение каждого типа имеет свой приоритет (от lowest до highest). Логические сообщения используются для информирования о нормальном ходе работы приложения (например, получен запрос для обработки, создан агент для обработки запроса и т.д.). Сообщения об ошибках используются для информирования о не нормальных, исключительных событиях в приложении.

Идея разделения сообщений по типам с независимыми приоритетами в каждом из типов происходит от желания создать инструмент, позволяющий получать логируемые сообщения на удаленном компьютере в режиме реального времени. Библиотека *so_log* специально разработана так, чтобы использовать глобального агента для рассылки сообщений во внешний мир. Благодаря этому внешний мониторинговый инструмент может подключиться к SObjectizer-приложению и получать логируемые сообщениях через SOP. Но наблюдатель, который использует подобный инструмент, может столкнуться с огромным потоком сообщений (десятки, а то и сотни, в секунду). Разделение сообщений на логические и сообщения об ошибках позволяет назначать при отображении минимальный интересующий наблюдателя приоритет для каждого типа сообщений. Например, приоритет high для логических (т.е. для важных сообщений о нормальном ходе работы) и medium для сообщений об ошибках (т.е. отбросить сообщения, которые являются всего лишь предупреждениями).

Кроме распространения логируемых сообщений во внешний мир посредством глобального агента *so_log* позволяет сохранять сообщениях в log-файлах (на данный момент поддерживаются суточные журнальные файлы и пятнадцатиминутные журнальные файлы) и/или отображать их на стандартные потоки вывода (в этом случае можно настраивать поток и вид сообщения для каждого из типов, что позволяет, например, направлять логические сообщения в *std::cout*, а сообщения об ошибках — в *std::cerr*).

2.4 Alternative Channel

При использовании штатных средств SObjectizer для организации клиентских (исходящих) коммуникационных каналов возникает небольшая проблема в случае, когда нужно уметь выбирать один из нескольких удаленных узлов. Например, некоторый компонент распределенной системы может иметь несколько точек входа: одну основную и некоторое число резервных. Когда соединение с основной точкой входа по каким-то причинам разрывается, нужно попробовать установить соединение с первой из резервных точек входа. Если это не удалось, то необходимо попробовать следующую и т.д.

Библиотека *so_alt_channel* предоставляет готовые средства для реализации подобного перебора альтернативных подключений.

2.5 Message Box API

Библиотека *mbapi* предоставляет SObjectizer-приложениям еще один способ обмена сообщениями, способными пересекать границу процесса. При использовании для построения распределенного приложения сообщений глобальных агентов возникает ряд неприятных моментов. Например, сообщения глобального агента рассылаются во все коммуникационные каналы, для которых доставка сообщения разрешена. Т.е. сообщение будет уходить даже в те коммуникационные каналы, на другой стороне которых никто не заинтересован в получении этих сообщений. Аналогично, если два приложения хотят организовать между собой *peer-to-peer* взаимодействие, то они должны отслеживать идентификаторы каналов, через которые они связанны между собой.

В противном случае широковещательная рассылка сообщений глобальных агентов будет доставлять сообщения и в другие части распределенного приложения, что может быть не желательно.

Идея *mbapi* в том, что *mbapi*-сообщения пересылаются от одного почтового ящика (*mbox*) к другому. В каждом приложении (каждой части распределенного приложения) декларируется, какие почтовые ящики существуют в этом приложении. Так же в каждом приложении запускается служба маршрутизации *mbapi*. Эта служба отслеживает состояния коммуникационных каналов и поддерживает таблицу маршрутизации, в которой сохраняется информация о том, через какие *mbox*-ы через коммуникационные каналы доступны. При отсылке *mbapi*-сообщения эта служба находит конкретный коммуникационный канал, через который доступен *mbox* получателя, и направляет сообщение в данный канал. Благодаря службе маршрутизации *mbapi* прикладным агентам в приложении не нужно думать об контроле состояний коммуникационных каналов. Для peer-to-peer взаимодействия им нужно знать только имена *mbox*-ов своих собеседников.

Еще одной отличительной чертой *mbapi* является то, что *mbapi*-сообщения не являются сообщениями глобальных агентов. *mbapi*-сообщение — это экземпляр объекта, класс которого производен от специального базового класса *mbapi_3::msg_t*. Это позволяет приложениям легко добавлять в свои протоколы новые типы сообщений без необходимости объявлять новых глобальных агентов или расширять существующие глобальные агенты новыми сообщениями. Все *mbapi*-сообщения являются сериализуемыми с помощью ObjESSty [OESS] объектами. Библиотека *mbapi* использует сериализацию для передачи *mbapi*-сообщений между процессами с помощью собственного глобального агента. Так же сериализуемость *mbapi*-сообщений может использоваться приложениями, например, для сохранения поступающих сообщений в БД для последующей обработки.

Поскольку *mbapi*-сообщения не являются обычными SObjectizer-сообщениями, для их получения агентам нужно выполнить дополнительное действие — объявить специального объекта-почтальона, который будет находить интересующие агента *mbapi*-сообщения и доставлять их агенту уже в виде обычных SObjectizer-сообщений. При этом объекты-почтальоны могут выделять *mbapi*-сообщения по разным критериям (например, по имени *mbox*-а получателя или отправителя, по типу *mbapi*-сообщения, по собственному, специфическому для агента критерию). Так же почтальон может перехватить *mbapi*-сообщение, т.е. запретить передачу *mbapi*-сообщения другим почтальонам, что позволяет использовать схемы работы, не доступные через обычные механизмы SObjectizer. Например, в приложение может быть встроена служба кэширования ответов. Она может перехватывать запросы клиента и проверять, выполнялся ли недавно подобный запрос и, если ответ на этот запрос есть в кэше, то отдавать клиенту ответ от кэша. Если же запроса в кэше нет, то запрос клиента перемаршрутизируется оригинальному исполнителю запросов. Причем исполнитель может даже не подозревать о существовании кэша, для него поток сообщений от клиента может выглядеть совершенно обычным. Такой механизм перехвата и перемаршрутизации *mbapi*-сообщений позволяет строить целые каскады обработчиков. Так, между кэшем и исполнителем может быть установлен дополнительный компонент, который будет контролировать количество запросов клиента, поступающих в единицу времени или выполнять какую-то промежуточную обработку сообщений (например, преобразовывать текстовые строки внутри сообщения из одной кодовой страницы в другую).

Часть II

SObjectizer System Configurator

Глава 3

Основные понятия

3.1 Подсистема so_sysconf

Подсистема *so_sysconf* является одной из самых ранних библиотек для SObjectizer. Ее появление было обусловлено наблюдением за разработкой первых реальных систем на основе SObjectizer. Пожалуй, наиболее характерной чертой таких систем было то, что связи между агентами устанавливались не во время компиляции (*compile-time*), а во время работы приложения (*run-time*). Даже если количество и состав агентов в приложении был фиксированным и известным на этапе компиляции, их реальное соединение (т.е. подписка на сообщения друг друга) выполнялась в run-time. Еще одной особенностью было то, что взаимодействующие агенты не нуждались в знании точных интерфейсов друг друга, т.е. им не нужно было на этапе компиляции видеть описания классов агентов. Поскольку агенты общаются через обмен сообщениями, то все что нужно видеть в compile-time — это описания классов сообщений, но эти описания выгодно было располагать отдельно от описания класса агента-владельца. Действительно, если есть классы агентов *A* и *B*, то их описания могут содержаться в файлах `a.hpp` и `b.hpp`, в то время, как описания их сообщений могут содержаться в `a_messages.hpp` и `b_messages.hpp`. А это означает, что, например, класс *A* может претерпевать очень серьезные изменения в процессе разработки и сопровождения (соответственно, все это будет отражаться в `a.hpp`), но вот состав его сообщений может быть гораздо стабильнее, поэтому все, кто использует только `a_messages.hpp` об происходящих изменениях могут даже не догадываться.

Другой характерной чертой первых систем на основе SObjectizer стало то, что они состояли из нескольких типов агентов, каждый из которых отвечал за выполнение какой-то одной частной задачи. А общая цель приложения достигалась за счет коммутации разнотипных агентов. При этом часть агентов из одного приложения вполне могла быть использована в другом приложении. Например, универсальные повторно-используемые агенты из подсистемы *so_log* или даже специфические прикладные агенты, которые можно было использовать в схожих прикладных задачах.

После выявления и анализа описанных выше особенностей возникла идея о том, что SObjectizer может создать условия, в которых приложения могут собираться из отдельных частей (подсистем или библиотек агентов) как из конструктора. Причем собираться в run-time, а не в compile-time. Центральное место в этой идее занимала возможность динамической загрузки и выгрузки динамически загружаемых библиотек современных операционных системах (*Dynamic Link Libraries (DLL)* в Windows и OS/2, *Shared Objects (SO)* в Unix). Подсистема *so_sysconf* является результатом реализации данной идеи.

Библиотека *so_sysconf* предоставляет приложениям возможность динамической загрузки DLL

в run-time, динамической регистрации и дерегистрации кооперaций, содержащихся в DLL, выгрузки DLL по необходимости. Использующие *so_sysconf* приложения, в большинстве случаев, состоят из небольшого загрузчика и нескольких DLL с реализацией специфической прикладной логики. Остальные DLL повторно используются из других проектов (например, таких как *so_log*, *gemont* и *mbapi*). А загрузчик представляет из себя небольшую программу, которая стартует SObjectizer и подсистему *so_sysconf*, после чего указывает *so_sysconf* имя конфигурационного файла с описанием состава приложения, и ожидает завершения работы SObjectizer. Основную часть работы по формированию приложения выполняет *so_sysconf* во время разбора и обработки конфигурационного файла. В последнее время приложения даже не создают собственных загрузчиков, а используют готовые универсальные загрузчики из состава *so_sysconf* (подробнее см. 5 на стр. 32).

По своей функциональности *so_sysconf* является аналогом фреймворка System Configurator из состава библиотеки ACE [ACE]. Но *so_sysconf* появилась гораздо раньше, чем ACE начала использоваться в SObjectizer, и ориентирована *so_sysconf* на кооперации агентов. Поэтому в настоящее время *so_sysconf* никак не использует System Configurator и является самостоятельной подсистемой.

3.2 DLL

DLL — это динамически загружаемая библиотека (`.dll` файл под Windows или `.so` файл под Unix) в которой находится код агентов и несколько *coop_handler* и/или *coop_factory*.

Каждая библиотека, которая предназначена для использования посредством *so_sysconf* должна иметь уникальный внутренний псевдоним (*alias*). Этот псевдоним необходим для идентификации библиотеки среди загруженных *so_sysconf* библиотек. Псевдоним задается разработчиком DLL и сообщается пользователям DLL в документации. Когда пользователь указывает *so_sysconf* на необходимость загрузки DLL, он должен сообщить *so_sysconf* этот псевдоним.

Такая схема с наличием у DLL псевдонима выглядит не очень удобной, однако, она позволяет абстрагироваться от физического имени DLL, которое под разными операционными системами может быть разным.

3.3 coop_handler

В некоторых случаях требуется, чтобы в SObjectizer-приложении была одна кооперация с конкретным именем, отвечающая за одну конкретную задачу. Например, такими кооперациями являются: сама подсистема *so_sysconf*, подсистема *so_log*, подсистема *gemont*. Имя кооперации заранее известно, нужно только управлять регистрацией и дерегистрацией данной кооперации. Для этого в DLL создается объект *coop_handler*, который знает имя кооперации и отвечает за ее регистрацию и дерегистрацию. Таким образом, *coop_handler* — это объект, который работает только с одной кооперацией.

Coop_handler выполняет две операции: регистрацию и дерегистрацию кооперации. При регистрации *coop_handler* может быть передано имя конфигурационного файла, в котором находятся параметры, необходимые кооперации. *Coop_handler* отвечает за чтение этого конфигурационного файла.

3.4 coop_factory

В противоположность *coop_handler* объект *coop_factory* является фабрикой однотипных кооперaций. Он предназначен для случаев, когда заранее не известно, сколько кооперaций требу-

ется создать в приложении и какие имена будут иметь эти кооперации.

Как и *coop_handler*, *coop_factory* отвечает за выполнение двух операций: регистрацию и дерегистрацию кооперации. Но, поскольку имена кооперации заранее не известны, эти имена должны передаваться в *coop_factory* при выполнении каждой операции. Так же, как и *coop_handler*, при регистрации кооперации *coop_factory* может получать имя конфигурационного файла с параметрами, необходимыми для работы кооперации.

Глава 4

Принципы работы

4.1 Общий принцип работы

Для использования *so_sysconf* необходимо зарегистрировать основную кооперацию подсистемы *so_sysconf* в уже запущенном SObjectizer Run-Time. Для чего предназначена функция *so_sysconf_2::register_coop()*. После этого подсистема *so_sysconf* работает до завершения SObjectizer Run-Time.

Основным интерфейсным агентом подсистемы sysconf является глобальный агент типа *so_sysconf_2::a_sysconf_t*, имеющий имя *so_sysconf_2::a_sysconf_t::agent_name()*. Осуществляя широковещательную рассылку сообщений этого агента можно загружать/выгружать DLL библиотеки, регистрировать и дерегистрировать находящиеся в них кооперации. Так, отсылка сообщения *a_sysconf_t::msg_load_dll* предписывает *so_sysconf* загрузить указанную DLL, а сообщение *a_sysconf_t::msg_unload_dll* — выгрузить DLL. Сообщения *a_sysconf_t::msg_reg_coop* и *a_sysconf_t::msg_make_coop* предназначены для регистрации кооперации через *coop_handler* и создание кооперации через *coop_factory*. Сообщение *a_sysconf_t::msg_dereg_coop* предназначено для дерегистрации ранее зарегистрированной кооперации.

Когда *so_sysconf* получает сообщение `msg_load_dll` (в котором содержится имя файла DLL и имя псевдонима DLL), он загружает указанную библиотеку (при условии, что библиотека с указанным псевдонимом не была загружена ранее). В этот момент в библиотеке начинают отрабатывать конструкторы глобальных переменных *coop_handler* и *coop_factory*. Эти конструкторы передают в *so_sysconf* информацию о том, какие кооперации и фабрики кооперации доступны в данной DLL (при этом используется псевдоним DLL). Таким образом *so_sysconf* узнает о существовании кооперации и фабрик кооперации. Соответственно, при выгрузке DLL (после получения сообщения `msg_unload_dll`) деструкторы *coop_handler* и *coop_factory* вычеркивают имена кооперации и фабрик из подсистемы *so_sysconf*.

При получении `msg_reg_coop` (имя кооперации и имя конфигурационного файла для кооперации) *so_sysconf* ищет в своем списке *coop_handler* с указанным именем и, если *coop_handler* найден, вызывает у него метод *coop_handler_t::reg()*, куда передает имя конфигурационного файла. Если этот метод возвращает *true*, то *so_sysconf* считает, что кооперация успешно зарегистрирована.

Аналогичные действия *so_sysconf* выполняет при получении `msg_make_coop`, но только ищет в своем списке не *coop_handler*, а *coop_factory*.

Когда кооперация создана и зарегистрирована *so_sysconf* увеличивает счетчик ссылок на DLL из которой кооперация была создана. Если *so_sysconf* получает сообщение `msg_unload_dll`, но счетчик ссылок на эту DLL отличен от нуля, то *so_sysconf* не выполняет выгрузку DLL.

При получении `msg_dereg_coop` с именем подлежащей дерегистрации кооперации *so_sysconf* проверяет, зарегистрирована ли кооперация в данный момент. Если зарегистрирована, то *so_sysconf* дерегистрирует ее, а при получении от SObjectizer подтверждения о дерегистрации кооперации уменьшает счетчик ссылок на DLL из которой кооперация была зарегистрирована.

4.1.1 Ручная работа с so_sysconf

Кооперацию подсистемы *so_sysconf* необходимо регистрировать при уже запущенном SObjectizer Run-Time. В штатном загрузчике `so_sysconf_process` (5.1 на стр. 32) для этого используется специальный агент `a_starter_t` с единственным событием `evt_start`:

```
void
a_starter_t::evt_start()
  {
    // Запускаем кооперацию sysconf-а.
    so_sysconf_2::register_coop(
        m_args.use_ostream_logger() ?
            new so_sysconf_2::ostream_sysconf_logger_t() : 0 );

    // Запускаем первоначальный скрипт. Если это не получится,
    // то завершим свою работу.
    if( !so_sysconf_2::run_script( m_args.initial_script() ) )
      {
        std::cerr << "unable to run initial script: "
            << m_args.initial_script() << std::endl;
        m_successful_start = false;

        so_4::api::send_msg( so_4::rt::sobjectizer_agent_name(),
            "msg_normal_shutdown", 0 );
      }
  }
```

Для того, чтобы после регистрации *so_sysconf* загрузить DLL необходимо отослать сообщение `msg_load_dll`:

```
so_4::api::send_msg_safely(
  so_sysconf_2::a_sysconf_t::agent_name(),
  "msg_load_dll",
  new so_sysconf_2::a_sysconf_t::msg_load_dll(
    // Имя файла DLL. Должно задаваться конкретное имя с учетом
    // особенностей конкретной ОС.
    "./sample.dll",
    // Уникальный псевдоним для DLL.
    "sample::dll" ) );
```

Регистрация кооперации через *coop_handler* осуществляется посредством сообщения *msg_reg_coop*:

```
so_4::api::send_msg_safely(
  so_sysconf_2::a_sysconf_t::agent_name(),
  "msg_reg_coop",
  new so_sysconf_2::a_sysconf_t::msg_reg_coop(
    // Имя обработчика кооперации.
    "sample::dll::main_coop",
    // Имя конфигурационного файла для кооперации.
    "etc/sample/main.cfg" ) );
```

Аналогичным образом выглядит создание кооперации через *coop_factory*:

```
so_4::api::send_msg_safely(
  so_sysconf_2::a_sysconf_t::agent_name(),
  "msg_make_coop",
  new so_sysconf_2::a_sysconf_t::msg_make_coop(
    // Имя фабрики кооперации.
    "sample::dll::some_factory",
    // Имя создаваемой кооперации.
    "sample::dll::coop_1",
    // Имя конфигурационного файла для кооперации.
    "etc/sample/child/params.cfg" ) );
```

4.1.2 Использование конфигурационного файла

Ручная работа с *so_sysconf* является весьма сложной. Гораздо удобнее описать конфигурацию приложения в конфигурационном файле и передать имя этого конфигурационного файла в *so_sysconf*. Подсистема *so_sysconf* сама произведет разбор конфигурационного файла и преобразует содержащиеся в нем команды в серию сообщений `msg_load_dll`, `msg_reg_coop` и `msg_make_coop`.

Вот пример конфигурационного файла из реального SObjectizer-приложения:

```
||
|| Скрипт начальной инициализации AAG 3
||
{sysconf-script

  |#
    Загрузка и инициализация средств обработки прерывания приложения.
    Содержит кооперации:
      so_sysconf_2::breakflag_handler::user_break_handler
      so_sysconf_2::breakflag_handler::system_break_handler
  #|
  {load-dll "so_sysconf.breakflag_handler"
    {alias  "so_sysconf_2::breakflag_handler"}
    {os-name-convert  "simple" }
  }
  || Обработчик прерывания приложения операционной системой.
  {reg-coop "so_sysconf_2::breakflag_handler::system_break_handler"
  }
  || Обработчик прерывания приложения пользователем.
  {reg-coop "so_sysconf_2::breakflag_handler::user_break_handler"
  }

  |#
    Подсистема so_log_1.
  #|
  {load-dll "so_sysconf_log.sysconf"
    {alias  "so_sysconf_log_1::sysconf" }
    {os-name-convert  "simple" }
  }
  {reg-coop "so_sysconf_log_1::sysconf::log" }

  |#
```

```
33    Точка входа в приложение по TCP/IP.
34    Фабрика: so_sysconf_2::ichannel::factory
35  #|
36  {load-dll "so_sysconf.ichannel.sysconf"
37    {alias  "so_sysconf_2::ichannel" }
38    {os-name-convert  "simple" }
39  }
40  {make-coop
41    {factory  "so_sysconf_2::ichannel::factory" }
42    {coop "so_sysconf_2::ichannel::tcp_entry"}
43    {cfg-file "ichannel.cfg" }
44  }
45
46  |#
47    Средства ретрансляции мониторинговой информации.
48    Содержат кооперацию: gemont_1::retranslator
49  #|
50  {load-dll "gemont.retranslator.sysconf"
51    {os-name-convert  "simple"}
52    {alias  "gemont_1::retranslator::sysconf"}
53  }
54  {reg-coop "gemont_1::retranslator" }
55
56  |#
57    Средства мониторинга текущего состояния системы.
58  #|
59  {load-dll "gemont.snapshot.sysconf"
60    {os-name-convert  "simple"}
61    {alias  "gemont_1::snapshot::sysconf"}
62  }
63  {make-coop
64    {factory  "gemont_1::snapshot::sysconf" }
65    {coop "gemont_1::snapshot::local" }
66    {cfg-file "etc/gemont_1/snapshot/local.cfg" }
67  }
68
69  |#
70    Средства мониторинга подсистемы sysconf через
71    средства gemont.
72  #|
73  {load-dll "so_sysconf_gemont.sysconf"
74    {alias  "so_sysconf_gemont_dll" }
75    {os-name-convert  "simple" }
76  }
77
78  {reg-coop "so_sysconf_gemont::sysconf_info_monitor"
79  }
80
81  |#
82    Загрузка и инициализация средств мониторинга доступных mbox-ов.
83    Содержит кооперацию:
84      mbapi_3_mbox::gemont
85  #|
86  {load-dll "mbapi.mbox.gemont.sysconf"
```

```
87    {os-name-convert  "simple" }
88    {alias  "mbapi_3_mbox::gemont"}
89  }
90  {reg-coop "mbapi_3_mbox::gemont"
91  }
92
93  |#
94    Загрузка и инициализация маршрутизатора MBAPI MBOX-сообщений.
95    Содержит кооперацию:
96      mbapi_3_mbox::core
97  #|
98  {load-dll "mbapi.mbox.core.sysconf"
99    {os-name-convert  "simple" }
100   {alias  "mbapi_3_mbox::core"}
101 }
102 {reg-coop "mbapi_3_mbox::core"
103   {cfg-file "mbapi_3_mbox/routers.cfg" }
104 }
105
106 |#
107   Загрузка workaround-а для сбора статистики о результатах
108   send-транзакций.
109   Псевдоним:
110     aag_3::workaround::send::result_dist
111   Содержит фабрику:
112     aag_3::workaround::send::result_dist::factory
113 #|
114 {load-dll "aag_3.workaround.send.result_dist"
115   {os-name-convert  "simple" }
116   {alias  "aag_3::workaround::send::result_dist"}
117 }
118
119 || Инициализация workaround-а для сбора статистики по send-result-ам,
120 || отсылаемым на smsc_map.
121 {make-coop
122   {factory "aag_3::workaround::send::result_dist::factory" }
123   {coop "aag_3::workaround::send::result_dist::default" }
124   {cfg-file "aag_3/workaround/send.result_dist/smsc_map.default.cfg"}
125 }
126
127 |#
128   Загрузка подсистемы smsc_map из AAG 3.
129   Содержит фабрику:
130     aag_3::smsc_map::factory
131 #|
132 {load-dll "aag.smsc_map"
133   {os-name-convert  "simple" }
134   {alias  "aag_3::smsc_map"}
135 }
136
137 || Инициализация smsc_map.
138 {make-coop
139   {factory "aag_3::smsc_map::factory" }
140   {coop "aag_3::smsc_map::default" }
```

```
141     {cfg-file "aag_3/smsc_map/default.cfg"}
142   }
143 }
```

Данный фрагмен показывает, что команды в конфигурационном файле являются аналогами соответствующих сообщений.

В теге `{load-dll}` используется специальный вспомогательный тег `{os-name-convert}`. Если он указан и имеет значение `simple`, то имя DLL преобразуется к соглашениям текущей ОС. Так, имя `aag.smsc_map` под Windows будет преобразовано в `aag.smsc_map.dll`, а под Unix в `libaag.smsc_map.so`.

Передача конфигурационного файла в *so_sysconf* выполняется с помощью функции *so_sysconf_2::run_script()*:

```
1 if( !so_sysconf_2::run_script( "etc/sysconf.cfg" ) )
2   // Загрузить конфигурационный файл не удалось!
3   ...
```

4.2 Агент a_shutdowner

Штатным способом корректного завершения работы SObjectizer в обычном приложении является отсылка сообщения `msg_normal_shutdown` агента `a_sobjectizer` после того, как все основные действия приложения завершены. Но, если приложение строится на основе подсистемы *so_sysconf* то отсылка `msg_normal_shutdown` не является хорошим решением, поскольку невозможно согласовать с его помощью одновременное завершение разных подсистем приложения. Может оказаться, что какой-то кооперации требуется много времени на то, чтобы завершить свою работу и корректно закрыть имеющиеся у него ресурсы (например, обменяться специальными сообщениями с удаленным приложением). Для того, чтобы работающие в *so_sysconf* кооперации могли выполнять согласованное завершение работы предназначен агент `a_shutdowner`.

Кооперация, которая нуждается в уведомлении о том, что приложению нужно завершить свою работу, требуется отослать сообщение `msg_register` агента *a_shutdowner_t::agent_name()* и передать в сообщении имя одного из своих агентов. Когда будет инициирована операция завершения работы приложения этому агенту будет отослано сообщение `msg_shutdown_started`. В ответ на него зарегистрировавшийся у `a_shutdowner` агент должен начать завершение своей работы. Когда кооперация может быть безопасно дерегистрирована она должна отослать сообщение `msg_deregister` с именем того же агента, что и в сообщении `msg_register`.

Для завершения работы SObjectizer-приложения в котором используется *so_sysconf* необходимо отослать сообщение `msg_shutdown` агента *a_shutdowner_t::agent_name()*. Получив его `a_shutdowner` рассылает сообщение `msg_shutdown_started` все зарегистрировавшимся у него агентам. После чего ожидает от них `msg_deregister`. После того, как все зарегистрировавшиеся агенты дерегистрируются при помощи `msg_deregister` агент `a_shutdowner` сам завершит работу SObjectizer через `msg_normal_shutdown`.

4.3 Агент a_trouble

Агент `a_trouble` предназначен для предоставления агентам возможности проинформировать *so_sysconf* о возникновении фатальной ошибки из-за которой все приложение не может продолжать свою работу и должно быть завершено. Этот агент входит в состав кооперации *so_sysconf* и регистрируется автоматически вместе с подсистемой *so_sysconf*. Агент `a_trouble` является владельцем сообщения `msg_fatal_error`. Когда это сообщение кем-то отсылается,

агент `a_trouble` получает его и инициирует завершение приложения путем отсылки сообщения `msg_shutdown` агента `a_shutdowner` (см. 4.2 на стр. 19).

Логика использования агента `a_trouble` проста. Как правило, в приложении есть несколько ключевых агентов выход из строя которых означает нарушение работоспособности приложения, т.е. возникновение проблемы. Одной из самых тривиальных и распространенных способов преодоления проблем является завершение приложения с тем, чтобы его можно было перезапустить после устранения причины возникновения проблемы. Именно эта практика поддерживается с помощью агента `a_trouble` — как только какой-нибудь важный агент понимает, что в сложившихся условиях продолжать работу нельзя, он генерирует сообщение `a_trouble.msg_fatal_error` после чего приложение закрывается. Далее в работу вмешивается либо оператор, который вручную перезапускает приложение, либо какая-нибудь автоматическая система рестарта приложений. Но важно, что `a_trouble` дает возможность SObjectizer-приложению, построенному на основе *so_sysconf*, корректно завершиться.

Как показывает практика, наибольшее количество фатальных ошибок генерируют *coop_handler* и *coop_factory* при невозможности разобрать конфигурационный файл кооперации.

Для отсылки сообщения `a_trouble.msg_fatal_error` предназначен вспомогательный статический метод класса *so_sysconf_2::a_troble_t*:

```
1 static void
2 send_msg_fatal_error(
3   //! Имя агента, который диагностировал критическую ошибку.
4   std::string agent,
5   //! Краткое имя ошибки.
6   std::string error_name,
7   //! Описание критической ошибки.
8   std::string desc );
```

Здесь параметр *agent* содержит имя агента, диагностировавшего ошибку (или имя кооперации, если ошибка диагностируется *coop_handler* и *coop_factory*). Параметр *error_name* содержит некоторое компактное, мнемоническое имя ошибки (например, *"config_file_not_found"* или *"coop_registration_failed"*), а параметр *desc* предназначен для подробного описания причины возникновения ошибки.

4.4 Агент a_breakflag_handler

В различных ОС есть несколько способов подать приложению сигнал о необходимости завершения работы по каким-либо причинам. Например, сигнал о нажатии пользователем `Ctrl+C` в консоли приложения или уведомление о том, что ОС начинает перезагрузку. SObjectizer не обрабатывает данные сигналы, оставляя реакцию на них на откуп приложению. В *so_sysconf* входят несколько агентов, облегчающих обработку подобных сигналов. Центральным из которых является агент `a_breakflag_handler`.

Подсистема *so_sysconf* различает два типа сигналов прерывания приложения:

- *so_sysconf_2::user_break*. Прерывание инициировано пользователем нажатием на `Ctrl+C`, `Ctrl+Break`. В Unix определяется возникновением сигнала *SIGINT*. В Win32 определяется возникновением сигналов *CTRL_C_EVENT*, *CTRL_BREAK_EVENT*. Прерывание приложения пользователем может быть проигнорировано приложением и приложение сможет нормально продолжить работу;
- *so_sysconf_2::system_break*. Прерывание инициировано операционной системой. В Unix определяется возникновением сигналов *SIGTERM*, *SIGHUP*. В Win32 определяется возникновением сигналов *CTRL_CLOSE_EVENT*, *CTRL_LOGOFF_EVENT*,

CTRL_SHUTDOWN_EVENT. Прерывание приложения операционной системой бесполезно игнорировать, т.к. это просто констатация свершившегося факта. Приложение может только постараться корректно завершить свою работу.

По-умолчанию, *so_sysconf* не перехватывает данные сигналы, что приводит к немедленному завершению приложения операционной системой. Если приложение вызывает *so_sysconf_2::setup_signal_handlers()*, то данные сигналы перехватываются и выставляются флаги, которые соответствуют типу прерывания приложения. Проверить установку соответствующего флага можно обратившись к *so_sysconf_2::is_set()*.

Обработчики сигналов прерывания приложения только перехватывают сигналы операционной системы и, кроме выставления флагов, не предпринимают никаких действий. Для типа сигнала *so_sysconf_2::user_break* это означает, что приложение просто не будет реагировать на `Ctrl+C`, `Ctrl+Break`.

Для того, чтобы по сигналу прерывания приложения инициировать корректное завершение приложения предназначена библиотека `so_sysconf.breakflag_handler`. Она имеет псевдоним `so_sysconf_2::breakflag_handler` и содержит две кооперации:

- `so_sysconf_2::breakflag_handler::user_break_handler`. Инициирует завершение приложения при возникновении сигнала типа *so_sysconf_2::user_break*;
- `so_sysconf_2::breakflag_handler::system_break_handler`. Инициирует завершение приложения при возникновении сигнала типа *so_sysconf_2::system_break*.

Для задействования данных обработчиков нужно загрузить библиотеку `so_sysconf.break_handler` и зарегистрировать кооперации `so_sysconf_2::breakflag_handler::user_break_handler` и/или `so_sysconf_2::breakflag_handler::system_break_handler`. Пример регистрации данных кооперaций через конфигурационный файл приведен в 4.1.2 на стр. 16.

Если какая-то кооперация не зарегистрирована, то сигналы соответствующих типов прерывания будут игнорироваться приложением.

4.5 Понятие app_paths

Опыт использования SObjectizer в разработке нескольких серверных приложений, работающих в режиме 24x7 показал, что файловая структура проинсталлированного приложения стремилась принять вид, подобный показанному ниже:

```
app_type/
+-- app/
    +-- bin/          # Здесь находятся различные версии
                      # exe и dll файлов приложения. Например:
        +-- 1.0.0/
        +-- 1.0.8/
        +-- 1.1.0/
    +-- component-1/  # Здесь находятся exe и dll файлы
                      # конкретного компонента. Отсюда
                      # происходит запуск этого компонента.
    +-- component-2/  # Аналогично для второго компонента и т.д.
+-- etc/              # Здесь располагаются конфигурационные
                      # файлы приложения.
    +-- component-1/  # Базовый каталог для всех конфигураций
                      # конкретного компонента.
```

```
        +-- 20060110/ # Одна версия конфигурации.
        +-- 20060118/ # Еще одна версия и т.д.
    +-- component-2/  # Аналогичное хранилище конфигурации для
                      # второго компонента.
+-- log/              # Здесь сохраняются логи приложения.
    +-- component-1/  # Логи первого компонента.
    +-- component-2/  # Логи второго компонента и т.д.
+-- data/             # Здесь располагаются различные файлы
                      # данных. Например, восстановочные базы
                      # данных, позволяющие приложению после
                      # сбоя восстановиться в наиболее близком
                      # к моменту сбоя состоянии.
    +-- component-1/
        +-- 1.0.0/    # Формат файлов данных может зависеть
                      # от версии поэтому для каждой версии
                      # создается свой подкаталог.
        +-- 1.0.8/
```

Под компонентом здесь понимается компонент приложения, который запускается в виде самостоятельного процесса и который взаимодействует с другими компонентами через какой-либо механизм IPC (Inter Process Communication). Практика показала, что организация SObjectizer-приложения в виде взаимодействующих самостоятельных процессов обеспечивает большую надежность, поскольку выход из строя одного компонента не влияет на остальные компоненты.

Например, предположим, что приложение обрабатывает транзакции, поступающие от нескольких клиентов по какому-то стандартному протоколу. В этом случае может быть выгодно для каждого клиента запустить собственный процесс-компонент. При этом окажется, что все компоненты имеют один тип и разделяют общую кодовую базу (т.е. один и тот же набор exe и dll файлов). В то же время, каждый компонент будет нуждаться в собственной копии конфигурационных файлов, log-файлов и вспомогательных файлов данных.

Описанная выше файловая структура обладает несколькими достоинствами:

- все файлы приложения собраны в одном месте, что облегчает контроль за их состоянием и их обновление;
- исходный набор exe и dll файлов находится в подкаталоге `app/bin/<version>`, откуда он копируется в подкаталог `app/component-N`. Запуск компонента производится из `app/component-N` и на некоторых операционных системах (например, Windows) модификация exe и dll файлов запущенного приложения становится невозможной. Следовательно, при необходимости обновления exe и dll файлов требуется остановить приложение. В случае приложений, работающих в режиме 24x7, этот останов/перезапуск должен занимать минимальное время. В случае указанной файловой структуры этого можно достичь следующим образом: в каталог `app/bin/<version>` загружаются обновленные версии exe/dll файлов. Для компонента создается новый каталог `app/component-N.k` куда копируются обновленные версии exe/dll. Затем компонент останавливается и рестартует, но уже из каталога `app/component-N.k`. Например, пусть есть компоненты alice и bob, а так же первая версия бинарных файлов 1.0.0. Файловая структура будет иметь вид:

  ```
  app_type/
  +-- app/
      +-- bin/
          +-- 1.0.0/
      +-- alice.1/
      +-- bob.1/
  ```

Затем потребовалось перейти на версию 1.0.8 для чего новая версия была загружена в каталог `app/bin/1.0.8`:

```
app_type/
+-- app/
    +-- bin/
        +-- 1.0.0/
        +-- 1.0.8/
    +-- alice.1/
    +-- bob.1/
```

Далее на основе версии 1.0.8 были созданы новые версии компонентов alice и bob:

```
app_type/
+-- app/
    +-- bin/
        +-- 1.0.0/
        +-- 1.0.8/
    +-- alice.1/
    +-- alice.2/
    +-- bob.1/
    +-- bob.2/
```

Теперь новые версии могут быть запущены из каталогов `app/alice.2` и `app/bob.2`. Достоинством здесь является и то, что в случае возникновения каких-либо проблем с новой версией ПО можно очень быстро откатиться к предыдущей версии просто рестартовав приложение из каталога с предыдущей версией;

- размещение бинарных файлов в отдельных подкаталогах для каждого компонента позволяет легко использовать разные версии ПО для компонентов. Например, для компонента bob может требоваться специализированная версия ПО, отличающаяся от стандартной версии, применяющейся для компонента alice.

- переход компонента на новую версию ПО может требовать модификации конфигурационных файлов компонента. Если модифицированные конфигурационные файлы будут располагаться отдельно от их предыдущей версии, то сохраняется возможность быстрого возврата к предыдущей версии ПО при возникновении каких-либо проблем с новой версией. Например, если для версии 1.0.0 конфигурационные файлы компонента alice располагались в подкаталоге `etc/alice/20060110`, то для версии 1.0.8 может быть выгодно разместить обновленные конфигурационные файлы в `etc/alice/20060118`. Аналогичные соображения актуальны и для файлов данных, размещаемых в `data/alice`;

- создавать новые подкаталоги для конфигурационных файлов может быть выгодно не только при переходе на новые версии ПО, но и при внесении серьезных изменений в текущую конфигурацию. Например, при необходимости сменить параметры подключения и аутентификации клиента. Сохранение неизменной предыдущей версии конфигурации позволяет легко вернуться на нее при возникновении проблем с новой конфигурацией.

Итак, при подходе, когда приложение стартует из одного каталога, а конфигурационные файлы, файлы данных и log-файлы размещены в других каталогах (имена которых могут изменяться от запуска к запуску) возникает задача определения расположения всех этих каталогов. Именно для решения этой задачи предназначено такое понятие, как *app_paths* — хранилище имен каталогов для размещения различных типов файлов приложение. Реализацией этого понятия в *so_sysconf* является класс *so_sysconf_2::app_paths_t*.

Предполагается, что в программе будет существовать один экземпляр класса *app_paths_t*, доступный через статический метод *app_paths_t::instance()*. Значения *app_paths* будут устанавливаться в момент старта приложения (штатные загрузчики получают их через параметры командной строки), а в приложении эти значения будут использоваться по необходимости. Например, *coop_handler* и *coop_factory* могут считать, что имена конфигурационных файлов задаются относительно значения *app_paths* и для определения расположения конфигурационного файла необходимо получить его полное имя с использованием соответствующего метода класса *app_paths_t*. Например, ниже приведен фрагмент кода *coop_factory* из реального SObjectizer-приложения:

```
bool
coop_factory_t::reg(
  const std::string & coop_name,
  const std::string & cfg_file,
  std::string & error_msg )
{
  cfg_t cfg;
  if( load_cfg_file(
      // Вот использование app_paths для получения полного
      // имени конфигурационного файла.
      so_sysconf_2::app_paths_t::instance()->etc_file_name( cfg_file ),
      cfg,
      error_msg ) )
  {
    std::auto_ptr< so_4::rt::agent_t > a_bufferizator(
        new a_bufferizator_t( coop_name + "::a_bufferizator", cfg ) );

    so_4::rt::agent_t * main_coop_agents[] =
    {
      a_bufferizator.release()
    };

    // Регистрируем кооперацию.
    so_4::rt::dyn_agent_coop_helper_t coop_helper(
      new so_4::rt::dyn_agent_coop_t(
        coop_name, main_coop_agents, 1 ) );
    if( !coop_helper.result() )
    {
      // Кооперация успешно зарегистрирована.
      so_log_1::logic[ so_log_1::normal ]
          [ so_log_1::a() << query_factory_name() ]
          [ so_log_1::n() << "factory::created" ]
          [ so_log_1::d() << "coop_name: " << coop_name
              << ", cfg_file: " << cfg_file ]();
      return true;
    }
    else
    {
      error_msg = std::string( "unable to register coop: " ) +
        coop_helper.result().m_desc;
    }
  }

  // Кооперацию зарегистрировать не удалось.
```

```
45   so_log_1::err[ so_log_1::highest ]
46       [ so_log_1::a() << query_factory_name() ]
47       [ so_log_1::n() << "factory::not_created"]
48       [ so_log_1::d() << "coop_name: " << coop_name
49           << ", cfg_file: " << cfg_file
50           << ", error: " << error_msg ]( __FILE__, __LINE__ );
51
52   // Это фатальная ошибка!
53   so_sysconf_2::a_trouble_t::send_msg_fatal_error(
54     query_factory_name(),
55     query_factory_name() + "::not_created",
56     error_msg );
57
58   return false;
59 }
```

4.6 Пример

В качестве примера реализации *coop_factory* можно рассмотреть фабрику `so_sysconf_2::ichannel::factory` штатной библиотеки для поддержки входящих коммуникационных каналов (см. 6.1 на стр. 39).

Заголовочный файл `so_sysconf_2_ichannel/h/coop_factory.hpp` описывает класс фабрики:

```
1  /*
2    SO SysConf 2
3  */
4
5  /*!
6    \since v.2.3.0
7    \file
8    \brief Фабрика кооперации точки входа в приложение.
9  */
10
11 #if !defined( SO_SYSCONF_2_ICHANNEL__COOP_FACTORY_HPP )
12 #define SO_SYSCONF_2_ICHANNEL__COOP_FACTORY_HPP
13
14 #include <so_sysconf_2/h/coop_factory.hpp>
15
16 namespace so_sysconf_2
17 {
18
19 namespace ichannel
20 {
21
22 //
23 // coop_factory_t
24 //
25
26 //! Фабрика коопераций.
27 /*!
28   Считывает конфигурацию из указанного файла и создает
29   кооперацию. Ошибка создания кооперации считается фатальной
```

```
30   ошибкой.
31
32   \par Псевдоним DLL:
33   so_sysconf_2::ichannel
34
35   \par Имя фабрики:
36   so_sysconf_2::ichannel::factory
37 */
38 class coop_factory_t :
39   public so_sysconf_2::coop_factory_t
40 {
41   typedef so_sysconf_2::coop_factory_t base_type_t;
42   public :
43     coop_factory_t();
44     virtual ~coop_factory_t();
45
46     //! Регистрирует кооперацию.
47     virtual bool
48     reg(
49       const std::string & coop_name,
50       const std::string & cfg_file,
51       std::string & error_msg );
52
53   private :
54     //! Псевдоним DLL.
55     static std::string  m_dll_alias;
56     //! Имя фабрики.
57     static std::string  m_factory_name;
58 };
59
60 } /* namespace ichannel */
61
62 } /* namespace so_sysconf_2 */
63
64 #endif
```

Класс *so_sysconf_2::ichannel::coop_factory_t* наследуется от базового класса *so_sysconf_2::ichannel*. Требуется переопределить виртуальный метод *reg()*, поскольку базовый класс не знает, как регистрировать кооперацию. Метод *dereg()* наследуется, т.к. для дерегистрации кооперации не нужно выполнять каких-то специфических действий, поэтому достаточно реализации *dereg()* из базового класса.

Файл реализации `so_sysconf_2_ichannel/coop_factory.cpp` содержит не только код класса *coop_factory_t*, но и код вспомогательного агента *a_failure_handler_t*:

```
1 /*
2   SO SysConf 2
3 */
4
5 /*!
6   \since v.2.3.0
7   \file
8   \brief Фабрика кооперации точки входа в приложение.
9 */
10
11 #include <so_sysconf_2_ichannel/h/coop_factory.hpp>
```

```
12
13 #include <memory>
14
15 #include <cpp_util_2/h/defs.hpp>
16 #include <cpp_util_2/h/lexcast.hpp>
17
18 #include <so_4/api/h/api.hpp>
19 #include <so_4/rt/h/rt.hpp>
20 #include <so_4/rt/h/msg_auto_ptr.hpp>
21 #include <so_4/rt/comm/h/a_sop_incoming_channel_processor.hpp>
22
23 #include <so_4/transport_layer/socket/h/pub.hpp>
24
25 #include <so_4/disp/active_obj/h/pub.hpp>
26 #include <so_4/disp/active_group/h/pub.hpp>
27
28 #include <so_sysconf_2/h/a_trouble.hpp>
29 #include <so_sysconf_2/h/app_paths.hpp>
30
31 #include <so_sysconf_2_ichannel/h/cfg.hpp>
32
33 namespace so_sysconf_2
34 {
35
36 namespace ichannel
37 {
38
39 //
40 // a_failure_handler_t
41 //
42
43 //! Обработчик неудачного создания серверного сокета.
44 /*!
45   Порождает фатальную ошибку, если создание серверного
46   сокета завершилось неудачно.
47 */
48 class a_failure_handler_t :
49   public so_4::rt::agent_t
50 {
51   public :
52     a_failure_handler_t(
53       //! Собственное имя.
54       const std::string & self_name,
55       /*! Имя агента-серверного сокета. */
56       const std::string & a_socksrv_name );
57     virtual ~a_failure_handler_t();
58
59     virtual const char *
60     so_query_type() const;
61
62     virtual void
63     so_on_subscription();
64
65     //! Реакция на неудачное создание коммуникационного сокета.
```

```
66      /*!
67        Порождается фатальная ошибка.
68
69        \par Приоритет:
70        0
71      */
72      void
73      evt_comm_socket_creation_fail(
74        const so_4::rt::comm::msg_fail & cmd );
75
76    private :
77      //! Псевдоним для базового типа.
78      typedef so_4::rt::agent_t base_type_t;
79
80      //! Имя агента-серверного сокета.
81      const std::string m_a_socksrv_name;
82  };
83
84  // Описание класса для SObjectizer-а
85  SOL4_CLASS_START( so_sysconf_2::ichannel::a_failure_handler_t )
86
87    SOL4_EVENT_STC(
88      evt_comm_socket_creation_fail,
89      so_4::rt::comm::msg_fail )
90
91    SOL4_STATE_START( st_initial )
92      SOL4_STATE_EVENT( evt_comm_socket_creation_fail )
93    SOL4_STATE_FINISH()
94
95  SOL4_CLASS_FINISH()
96
97  // Реализация класса
98  a_failure_handler_t::a_failure_handler_t(
99    const std::string & self_name,
100   const std::string & a_socksrv_name )
101   :
102     base_type_t( self_name.c_str() ),
103     m_a_socksrv_name( a_socksrv_name )
104 {
105 }
106
107 a_failure_handler_t::~a_failure_handler_t()
108 {
109 }
110
111 void
112 a_failure_handler_t::so_on_subscription()
113 {
114   so_subscribe(
115       "evt_comm_socket_creation_fail",
116       m_a_socksrv_name,
117       "msg_fail" );
118 }
119
```

```
120 void
121 a_failure_handler_t::evt_comm_socket_creation_fail(
122   const so_4::rt::comm::msg_fail & cmd )
123 {
124   // Это фатальная ошибка!
125   so_sysconf_2::a_trouble_t::send_msg_fatal_error(
126     so_query_name(),
127     "unable_to_create_server_socket",
128     cmd.m_reason );
129 }
130
131 //
132 // coop_factory_t
133 //
134
135 std::string
136 coop_factory_t::m_dll_alias( "so_sysconf_2::ichannel" );
137
138 std::string
139 coop_factory_t::m_factory_name( "so_sysconf_2::ichannel::factory" );
140
141 coop_factory_t::coop_factory_t()
142   :
143     base_type_t( m_dll_alias.c_str(), m_factory_name.c_str() )
144 {
145 }
146
147 coop_factory_t::~coop_factory_t()
148 {
149 }
150
151 bool
152 coop_factory_t::reg(
153   const std::string & coop_name,
154   const std::string & cfg_file,
155   std::string & error_msg )
156 {
157   cfg_t cfg;
158   if( load_cfg_file( app_paths_t::instance()->etc_file_name( cfg_file ),
159       cfg, error_msg ) )
160   {
161     // Имя агента-сокета в кооперации.
162     std::string a_socksrv_name( cfg.m_channel_agent_name );
163     if( a_socksrv_name.empty() )
164       a_socksrv_name = coop_name + "::a_ichannel";
165
166     // Коммуникационный сокет для входа в MBAPI Server.
167     using namespace so_4::rt::comm;
168     using namespace so_4::transport_layer;
169     using namespace so_4::transport_layer::socket;
170     std::auto_ptr< a_sop_incoming_channel_processor_t > a_socksrv(
171         new a_sop_incoming_channel_processor_t(
172             a_socksrv_name,
173             create_acceptor_controller(
```

```
174                 acceptor_params( cfg.m_ip ),
175                 cfg.m_channel_params ) ) );
176     a_socksrv->set_handshaking_params( cfg.m_handshaking_params );
177 
178     // Разберемся, на какой нити будет работать транспортный агент.
179     if( cfg.m_is_active_obj )
180       so_4::disp::active_obj::make_active( *a_socksrv );
181     else if( !cfg.m_active_group_name.empty() )
182       so_4::disp::active_group::make_member( *a_socksrv,
183           cfg.m_active_group_name );
184 
185     // Контроллер невозможности создания серверного сокета.
186     std::auto_ptr< so_4::rt::agent_t > a_failure_handler(
187         new a_failure_handler_t(
188             coop_name + "::a_failure_handler",
189             a_socksrv_name ) );
190 
191     so_4::rt::agent_t * main_coop_agents[] =
192     {
193       a_socksrv.release(),
194       a_failure_handler.release()
195     };
196 
197     // Регистрируем кооперацию.
198     so_4::rt::dyn_agent_coop_helper_t coop_helper(
199       new so_4::rt::dyn_agent_coop_t(
200         coop_name.c_str(), main_coop_agents,
201         CPP_UTIL_2_ASIZE( main_coop_agents ) ) );
202     if( !coop_helper.result() )
203       return true;
204 
205     error_msg = std::string( "unable to register coop: " ) +
206       cpp_util_2::slexcast( coop_helper.result() );
207   }
208 
209   // Это фатальная ошибка!
210   so_sysconf_2::a_trouble_t::send_msg_fatal_error(
211     "so_sysconf_2_ichannel::coop_factory_t::reg()",
212     "unable_to_register_coop",
213     error_msg );
214 
215   return false;
216 }
217 
218 //! Этот объект отвечает за фабрику.
219 coop_factory_t g_coop_factory;
220 
221 } /* namespace ichannel */
222 
223 } /* namespace so_sysconf_2 */
```

Агент `a_failure_handler` необходим для того, чтобы получить результат создания серверного сокета агентом типа *so_4::rt::comm::a_sop_incoming_channel_processor_t*. Этот результат становится известным только после регистрации кооперации и сообщается через сообщение

агента-канала. Поэтому результат не может быть просто получен в методе *coop_handler_t::reg()*. Этот результат приходит в виде события `evt_comm_socket_creation_fail` агента `a_failure_handler`.

Глава 5

Штатные загрузчики

Практика использования *so_sysconf* в различных SObjectizer-приложениях показала, что со временем практически вся прикладная логика приложений оказывается в DLL, собираемых вместе через *so_sysconf*. А функция *main()* приложения выполняет всего две операции: запуск SObjectizer и регистрацию подсистемы *so_sysconf* с последующей обработкой единственного конфигурационного файла. При этом реализация *main()* чаще всего просто тиражировалась из проекта в проект путем обычного копирования соответствующего cpp-файла. Для того, чтобы избежать этого копирования и предоставить всем заинтересованным приложениям унифицированный *so_sysconf*-загрузчик в состав *so_sysconf* включено два штатных загрузчика: `so_sysconf.process` (SObjectizer-приложение в виде обычного процесса) и `so_sysconf_ntservice` (SObjectizer-приложение в виде NT Service в операционных системах Windows).

5.1 so_sysconf.process

5.1.1 Назначение

Приложение `so_sysconf.process` отвечает за запуск SObjectizer с указанным диспетчером, за регистрацию подсистемы *so_sysconf* и за передачу подсистеме *so_sysconf* указанного конфигурационного файла для обработки. Приложение завершает свою работу после завершения SObjectizer.

5.1.2 Формат

```
so_sysconf.process [options]

-s, --script <file>     use file as initial sysconf script
    --disp-one-thread   use one thread SObjectizer dispatcher
    --disp-active-obj   use active objects SObjectizer dispatcher
    --disp-active-group use active group SObjectizer dispatcher
    --app-path-etc <path>       path for configuration files
    --app-path-log <path>       path for log files
    --app-path-data <path>      path for data files
    --app-path-tmp <path>       path for temporary files
    --ostream-logger    use ostream sysconf logger for sysconf
                        cooperation
```

```
-h, --help      show this help

Note: --disp-one-thread and --disp-active-obj cannot be used together
```

5.1.3 Описание

Параметры *–disp-one-thread*, *–disp-active-obj* и *–disp-active-group* указывают, какой диспетчер будет использоваться при старте SObjectizer. По умолчанию, если ни один из параметров не указан, будет использоваться диспетчер с активными объектами (что аналогично использованию только параметра *–disp-active-obj*). Если указан только *–disp-one-thread*, то используется диспетчер с одной рабочей нитью. Если указывается только *–disp-active-group*, то создается диспетчер с активными группами, а в качестве подчиненного ему диспетчера будет создан:

- диспетчер с одной рабочей нитью, если указан параметр *–disp-one-thread* (т.е. в командной строке должны одновременно присутствовать *–disp-one-thread* и *–disp-active-group*);
- диспетчер с активными объектами, если указан параметр *–disp-active-obj* (т.е. в командной строке есть и *–disp-active-obj* и *–disp-active-group*) или если больше ничего не указано (т.е. присутствует только *disp-active-group*).

Параметр *–script* (короткий аналог *-s*) задает имя конфигурационного файла, которое будет передано подсистеме *so_sysconf* после успешного старта SObjectizer. Если в процессе загрузки данного конфигурационного файла произойдет ошибка, то `so_sysconf.process` завершит свою работу. Это обязательный параметр, который должен быть задан в командной строке.

Параметры *–app-path-etc*, *–app-path-log*, *–app-path-data* и *–app-path-tmp* задают соответствующие части глобального *app_paths* для приложения (подробнее см. 4.5 на стр. 21). Это необязательные параметры, в случае отсутствия какого-либо из них в качестве соответствующего значения *app_paths* принимается `.` (текущий каталог).

Параметр *–ostream-logger* предписывает подсистеме *so_sysconf* отображать на стандартные потоки вывода сообщения о происходящих с *so_sysconf* событиях.

Например, следующая командная строка:

```
so_sysconf.process --disp-active-group --ostream-logger \
--app-path-etc etc --app-path-log log/aag_3 --app-path-data log/db \
--script etc/sysconf.cfg
```

предписывает запустить SObjectizer с использованием диспетчера с активными группами (в качестве подчиненного будет использоваться диспетчер с активными объектами) и передать подсистеме *so_sysconf* в качестве конфигурационного файла файл `etc/sysconf.cfg`. Все происходящие с *so_sysconf* события будут отображаться на стандартный поток вывода. Глобальный *app_paths* для приложения получит следующие значения:

- каталог `etc` в качестве пути к конфигурационным файлам;
- каталог `log/aag_3` в качестве пути для log-файлов;
- каталог `log/db` в качестве пути для файлов данных;
- текущий каталог в качестве пути для временных файлов (используется значение по умолчанию, поскольку параметр *–app-path-tmp* не задан).

В результате запуска и останова приложения на стандартный поток вывода могут быть отображены следующие сообщения:

```
load dll (alias: so_sysconf_2::breakflag_handler, file_name:
  so_sysconf.breakflag_handler.dll)
adding coop handler (coop_name: so_sysconf_2::breakflag_handler::
  user_break_handler, dll_alias: so_sysconf_2::breakflag_handler)
adding coop handler (coop_name: so_sysconf_2::breakflag_handler::
  system_break_handler, dll_alias: so_sysconf_2::breakflag_handler)
register coop (coop_name: so_sysconf_2::breakflag_handler::
  system_break_handler, cfg_file: )
register coop (coop_name: so_sysconf_2::breakflag_handler::
  user_break_handler, cfg_file: )
load dll (alias: so_sysconf_log_1::sysconf, file_name:
  so_sysconf_log.sysconf.dll)
adding coop handler (coop_name: so_sysconf_log_1::sysconf::log,
  dll_alias: so_sysconf_log_1::sysconf)
register coop (coop_name: so_sysconf_log_1::sysconf::log, cfg_file: )
load dll (alias: so_sysconf_2::ichannel, file_name:
  so_sysconf.ichannel.sysconf.dll)
adding coop factory (factory_name: so_sysconf_2::ichannel::factory,
  dll_alias: so_sysconf_2::ichannel)
make coop (factory_name: so_sysconf_2::ichannel::factory, coop_name:
  so_sysconf_2::ichannel::tcp_entry, cfg_file: ichannel.cfg)
load dll (alias: gemont_1::retranslator::sysconf, file_name:
  gemont.retranslator.sysconf.dll)
adding coop handler (coop_name: gemont_1::retranslator, dll_alias:
  gemont_1::retranslator::sysconf)
register coop (coop_name: gemont_1::retranslator, cfg_file: )
load dll (alias: gemont_1::snapshot::sysconf, file_name:
  gemont.snapshot.sysconf.dll)
adding coop factory (factory_name: gemont_1::snapshot::sysconf,
  dll_alias: gemont_1::snapshot::sysconf)
make coop (factory_name: gemont_1::snapshot::sysconf, coop_name:
  gemont_1::snapshot::local, cfg_file: etc/gemont_1/snapshot/local.cfg)
load dll (alias: so_sysconf_gemont_dll, file_name:
  so_sysconf_gemont.sysconf.dll)
adding coop handler (coop_name: so_sysconf_gemont::sysconf_info_monitor,
  dll_alias: so_sysconf_gemont_dll)
register coop (coop_name: so_sysconf_gemont::sysconf_info_monitor,
  cfg_file: )
load dll (alias: mbapi_3_mbox::gemont, file_name:
  mbapi.mbox.gemont.sysconf.dll)
adding coop handler (coop_name: mbapi_3_mbox::gemont, dll_alias:
  mbapi_3_mbox::gemont)
register coop (coop_name: mbapi_3_mbox::gemont, cfg_file: )
load dll (alias: mbapi_3_mbox::core, file_name:
  mbapi.mbox.core.sysconf.dll)
adding coop handler (coop_name: mbapi_3_mbox::core, dll_alias:
  mbapi_3_mbox::core)
register coop (coop_name: mbapi_3_mbox::core, cfg_file:
  mbapi_3_mbox/routers.cfg)
load dll (alias: aag_3::sms_hist_cls, file_name: aag.sms_hist_cls.dll)
adding coop handler (coop_name: aag_3::sms_hist_cls, dll_alias:
  aag_3::sms_hist_cls)
register coop (coop_name: aag_3::sms_hist_cls, cfg_file:
  aag_3/safe_sms_history.cfg)
```

```
load dll (alias: aag_3::workaround::send::result_dist,
  file_name: aag_3.workaround.send.result_dist.dll)
adding coop factory (factory_name: aag_3::workaround::send::
  result_dist::factory, dll_alias: aag_3::workaround::send::
  result_dist)
make coop (factory_name: aag_3::workaround::send::result_dist::
  factory, coop_name: aag_3::workaround::send::result_dist::
  default, cfg_file: aag_3/workaround/send.result_dist/
  smsc_map.default.cfg)
load dll (alias: aag_3::smsc_map, file_name: aag.smsc_map.dll)
adding coop factory (factory_name: aag_3::smsc_map::factory,
  dll_alias: aag_3::smsc_map)
make coop (factory_name: aag_3::smsc_map::factory, coop_name:
  aag_3::smsc_map::default, cfg_file: aag_3/smsc_map/default.cfg)
load dll (alias: aag_3::workaround::beeline::msgid, file_name:
  aag_3.workaround.beeline.msgid.dll)
adding coop factory (factory_name: aag_3::workaround::beeline::
  msgid::factory, dll_alias: aag_3::workaround::beeline::msgid)
make coop (factory_name: aag_3::workaround::beeline::msgid::
  factory, coop_name: aag_3::beeline::msgid::bserver.trx, cfg_file:
  aag_3/workaround/beeline/msgid/bserver.trx.cfg)
load dll (alias: aag_3::workaround::bercut_cpa_trx, file_name:
  aag_3.workaround.bercut_cpa_trx.dll)
adding coop factory (factory_name: aag_3::workaround::bercut_cpa_trx::
  factory, dll_alias: aag_3::workaround::bercut_cpa_trx)
make coop (factory_name: aag_3::workaround::bercut_cpa_trx::factory,
  coop_name: aag_3::bercut_cpa_trx::bserver.trx, cfg_file:
  aag_3/workaround/bercut_cpa_trx/bserver.trx.cfg)
load dll (alias: aag_3::workaround::send::bufferizator, file_name:
  aag_3.workaround.send.bufferizator.dll)
adding coop factory (factory_name: aag_3::workaround::send::
  bufferizator::factory, dll_alias: aag_3::workaround::send::
  bufferizator)
make coop (factory_name: aag_3::workaround::send::bufferizator::
  factory, coop_name: aag_3::send::bufferizator::local.trx, cfg_file:
  aag_3/workaround/send.bufferizator/local.trx.cfg)
make coop (factory_name: aag_3::workaround::send::bufferizator::
  factory, coop_name: aag_3::send::bufferizator::bserver.trx,
  cfg_file: aag_3/workaround/send.bufferizator/bserver.trx.cfg)
load dll (alias: aag_3::smpp_smsc, file_name: aag.smpp_smsc.dll)
adding coop factory (factory_name: aag_3::smpp_smsc::factory,
  dll_alias: aag_3::smpp_smsc)
make coop (factory_name: aag_3::smpp_smsc::factory, coop_name:
  aag_3::smpp_smsc::bserver.trx, cfg_file:
  aag_3/smpp_smsc/bserver.trx.cfg)
make coop (factory_name: aag_3::smpp_smsc::factory, coop_name:
  aag_3::smpp_smsc::local.trx, cfg_file:
  aag_3/smpp_smsc/local.trx.cfg)
load dll (alias: aag_3::smpp_entry, file_name: aag.smpp_entry.dll)
adding coop factory (factory_name: aag_3::smpp_entry::factory,
  dll_alias: aag_3::smpp_entry)
make coop (factory_name: aag_3::smpp_entry::factory, coop_name:
  aag_3::smpp_entry::local.trx, cfg_file:
  aag_3/smpp_entry/local.trx.cfg)
```

```
coop deregistered (coop_name: aag_3::smpp_smsc::local.trx)
coop deregistered (coop_name: aag_3::beeline::msgid::bserver.trx)
coop deregistered (coop_name: aag_3::bercut_cpa_trx::bserver.trx)
coop deregistered (coop_name: aag_3::send::bufferizator::bserver.trx)
coop deregistered (coop_name: aag_3::send::bufferizator::local.trx)
coop deregistered (coop_name: aag_3::smpp_entry::local.trx)
coop deregistered (coop_name: aag_3::smpp_smsc::bserver.trx)
coop deregistered (coop_name: aag_3::sms_hist_cls)
coop deregistered (coop_name: aag_3::smsc_map::default)
coop deregistered (coop_name: aag_3::workaround::send::result_dist::
  default)
coop deregistered (coop_name: gemont_1::retranslator)
coop deregistered (coop_name: gemont_1::snapshot::local)
coop deregistered (coop_name: mbapi_3_mbox::core)
coop deregistered (coop_name: mbapi_3_mbox::gemont)
coop deregistered (coop_name: so_sysconf_2::breakflag_handler::
  system_break_handler)
coop deregistered (coop_name: so_sysconf_2::breakflag_handler::
  user_break_handler)
coop deregistered (coop_name: so_sysconf_2::ichannel::tcp_entry)
coop deregistered (coop_name: so_sysconf_gemont::sysconf_info_monitor)
coop deregistered (coop_name: so_sysconf_log_1::sysconf::log)
```

5.2 so_sysconf.ntservice

5.2.1 Назначение

Приложение `so_sysconf.ntservice` выполняет ту же задачу, что и `so_sysconf.process`, но предназначено для того, чтобы работать в качестве Windows NT Services. Поэтому кроме возможностей `so_sysconf.process` предоставляет так же возможность инсталляции/деинсталляции, запуска/останова Windows сервиса.

5.2.2 Формат

```
so_sysconf.ntservice [options]

    --svc-name <NAME>   service name
    --svc-work-path <PATH>      use this path as current path for service
    --svc-install       install service
    --svc-use-current-path      use current path as work path when
                                install service
    --svc-remove        remove service
    --svc-start start service
    --svc-stop  stop (shutdown) service
    --svc-manual-startup        service must be started manualy
                                (default: auto)
    --svc-debug service must be in debug mode, as console application,
                not service (default: service)
-s, --script <file>     use file as initial sysconf script
    --disp-one-thread   use one thread SObjectizer dispatcher
    --disp-active-obj   use active objects SObjectizer dispatcher
    --disp-active-group use active group SObjectizer dispatcher
```

```
    --app-path-etc <path>       path for configuration files
    --app-path-log <path>       path for log files
    --app-path-data <path>      path for data files
    --app-path-tmp <path>       path for temporary files
    --ostream-logger    use ostream sysconf logger for sysconf
                        cooperation
-h, --help      show this help

Note: --disp-one-thread and --disp-active-obj cannot be used
      together

Note: --svc-install and --svc-remove cannot be used together

Note: --svc-work-path and --svc-use-current-path connot be used
      together
```

5.2.3 Описание

Параметр *–svc-name* задает имя сервиса в операционной системе (это имя, которое затем можно указывать в таких командах, как `net start` и `net stop`). Если этот параметр используется совместно с *–svc-install*, то он задает имя инсталлируемого сервиса. Если же *–svc-name* используется совместно с *svc-start*, *svc-stop* и *–svc-remove*, то он задает имя уже проинсталлированного сервиса над которым нужно выполнять определенное действие.

Параметр *–svc-install* указывает, что сервис нужно зарегистрировать. По умолчанию проинсталлированный сервис помечается как автоматически запускаемый при старте системы. Если требуется, чтобы новый сервис запускался вручную, то при его инсталляции необходимо указать параметр *–svc-manual-startup*. Параметр *–svc-work-path* при инсталляции сервиса указывает, какой каталог должен быть домашним для сервиса (т.е., какой каталог будет текущим для сервиса, когда сервис будет запущен операционной системой). Параметр *–svc-use-current-path* предписывает `so_sysconf.ntservice` использовать в качестве домашнего тот каталог, из которого `so_sysconf.ntservice` запускалась для инсталляции сервиса. Поэтому параметры *–svc-work-path* и *–svc-use-current-path* не могут использоваться совместно.

Параметр *–svc-start* предписывает запустить сервис, который должен был быть до этого проинсталлирован. А параметр *–svc-stop*, напротив, предписывает остановить сервис. Вместо запуска `so_sysconf.ntservice` с этими параметрами можно использовать `net start` и `net stop`.

Параметр *–svc-remove* предписывает деинсталлировать проинсталлированный ранее сервис.

Параметр *–svc-debug* указывает `so_sysconf.ntservice` не выполнять никаких действий с базой данных сервисов Windows, а запускать сервис в виде обычного консольного приложения. Этот режим очень удобен при отладке.

Остальные параметры имеют то же самое назначение, что и для `so_sysconf.process` (см. 5.1 на стр. 32).

Если параметр *–svc-debug* не указан, то все основные параметры (т.е. тип диспетчера и значения *app_paths*) должны указываться при инсталляции сервиса. Затем эти параметры сохраняются в описании сервиса в базе данных сервисов Windows и используются при последующих стартах сервиса. Поэтому, если необходимо запустить приведенный в разделе 5.1.3 пример в качестве сервиса, то необходимо выполнить следующие запуски `so_sysconf.ntservice`:

```
> so_sysconf.ntservice --svc-name MyService --svc-install \
  --svc-use-current-path \
  --disp-active-group --ostream-logger \
  --app-path-etc etc --app-path-log log/aag_3 \
```

```
  --app-path-data log/db \
  --script etc/sysconf.cfg

> so_sysconf.ntservice --svc-name MyService --svc-start
```

5.2.4 Проблемы

Для работы с базой данных сервисов используется функциональность, обеспечиваемая библиотекой ACE. К сожалению, в процессе работы с `so_sysconf.ntservice` были выявлены некоторые нерегулярные проблемы с дерегистрацией и остановом сервисов. Есть подозрения, что данные проблемы могут быть спровоцированы либо не вполне корректным кодом в ACE, либо не корректной работой со средствами ACE в `so_sysconf.ntservice`.

5.3 so_sysconf.daemon

5.3.1 Назначение

Приложение `so_sysconf.daemon` выполняет ту же задачу, что и `so_sysconf.process`, но предназначено для того, чтобы работать в качестве демона в Unix.

Приложение `so_sysconf.daemon` получает такой же набор аргументов, как и `so_sysconf.process`, но перед выполнением основной работы (старта SObjectizer и подсистемы *so_sysconf*):

- выполняется вызов fork;
- выполняется вызов setsid;
- текущим каталогом делается корневой каталог;

Примечание. Конкретные детали определяет реализация функции `ACE::daemonize()` из состава библиотеки ACE.

5.3.2 Формат

```
SO SysConf Process Launcher

so_sysconf.daemon [options]

-s, --script <file>     use file as initial sysconf script
    --disp-one-thread   use one thread SObjectizer dispatcher
    --disp-active-obj   use active objects SObjectizer dispatcher
    --disp-active-group use active group SObjectizer dispatcher
    --app-path-etc <path>       path for configuration files
    --app-path-log <path>       path for log files
    --app-path-data <path>      path for data files
    --app-path-tmp <path>       path for temporary files
    --ostream-logger    use ostream sysconf logger for sysconf cooperation
-h, --help      show this help

Note: --disp-one-thread and --disp-active-obj cannot be used together
```

Глава 6

Штатные кооперации коммуникационных каналов

Для реализации распределенных приложений средствами SObjectizer необходимо создавать в приложении один или несколько транспортных агентов, которые будут поддерживать коммуникационные каналы с другими приложениями. Поскольку *so_sysconf* делает возможным сборку приложения из готовых DLL как из конструктора, то возможность создавать через *so_sysconf* транспортных агентов так же повышает гибкость конструкции приложения. Отделение деталей организации транспорта сообщений от прикладной логики позволяет прозрачно переконфигурировать приложение, меняя виды транспорта и топологию (т.к. расположение прикладных агентов на разных узлах сети).

В настоящее время основным видом транспорта для SObjectizer является SOP-протокол поверх потоковых TCP/IP соединений. В связи с этим различаются два типа транспортных агентов: серверные (класс *so_4::rt::comm::a_sop_incoming_channel_processor_t*), которые отвечают за обслуживание серверных TCP/IP сокетов, и клиентские (класс *so_4::rt::comm::a_sop_outgoing_channel_t*), которые отвечают за обслуживание клиентских TCP/IP сокетов и подключение к серверным сокетам. Для работы через *so_sysconf* с этими типами агентов в состав *so_sysconf* включены две DLL:

- `so_sysconf.ichannel.sysconf` (псевдоним `so_sysconf_2::ichannel`), в которой находится фабрика `so_sysconf_2::ichannel::factory` для создания агентов типа *so_4::rt::comm::a_sop_incoming_channel_processor_t*;
- `so_sysconf.ochannel.sysconf` (псевдоним `so_sysconf_2::ochannel`), в которой находится фабрика `so_sysconf_2::ochannel::factory` для создания агентов типа *so_4::rt::comm::a_sop_outgoing_channel_t*.

Приложению, нуждающемуся в поддержке SOP-коммуникационных каналов достаточно использовать эти DLL и находящиеся в них *coop_factory* для создания необходимых транспортных агентов.

6.1 Входящий канал

Созданием агентов для серверных TCP/IP сокетов занимается библиотека `so_sysconf.ichannel.sysconf` (псевдоним `so_sysconf_2::ichannel`) и находя-

щаяся в ней фабрика `so_sysconf_2::ichannel::factory`. Для ее использования в конфигурационный файл *so_sysconf* необходимо добавить инструкции:

```
1 |#
2   Точка входа в приложение по TCP/IP.
3   Фабрика: so_sysconf_2::ichannel
4 #|
5 {load-dll "so_sysconf.ichannel.sysconf"
6   {alias  "so_sysconf_2::ichannel" }
7   {os-name-convert  "simple" }
8 }
9 {make-coop
10  {factory  "so_sysconf_2::ichannel::factory" }
11  {coop "so_sysconf_2::ichannel::tcp_entry"}
12  {cfg-file "ichannel.cfg" }
13 }
```

Тег `{make-coop {coop}}` задает имя кооперации, которая будет создана фабрикой. Тег `{make-coop {cfg-file}}` задает имя конфигурационного файла с параметрами нового соединения. Для фабрики `so_sysconf_2::ichannel::factory` имя конфигурационного файла должно быть задано обязательно.

Важно: имя конфигурационного файла должно задаваться относительно пути к конфигурационным файлам из *app_paths* (см. 4.5 на стр. 21). Например, если при старте приложения в аргументе *–app-path-etc* было задано имя `../etc/alice/20060110`, то имя `ichannel.cfg` будет преобразовано в `../etc/alice/20060110/ichannel.cfg`.

Конфигурационный файл для фабрики `so_sysconf_2::ichannel::factory` имеет формат:

```
{so_sysconf_ichannel
  {ip <str> }
  [{channel_agent_name <str>}]
  [{active_obj}]
  [{active_group_name <str>}]

  [{channel_params <параметры>}]
  [{handshaking_params <параметры>}]
}
```

Обязательным является только тег `{ip}`, он задает IP-адрес серверного сокета.

Тег `{channel_agent_name}` задает имя транспортного агента. По умолчанию имя транспортного агента формируется фабрикой на основе имени кооперации. Но, если транспортный агент должен получить конкретное имя (например, для того, чтобы можно было подписываться на его сообщения), то это имя можно задать посредством данного тега.

Если указан тег `{active_obj}`, то транспортный агент будет объявлен активными агентом (соответственно, требуется, чтобы приложение использовало диспетчер с активными агентами).

Если указан тег `{active_group_name}`, то он содержит имя активной группы, в которую должен входить транспортный агент (соответственно, требуется, чтобы приложение использовало диспетчер с активными группами).

Теги `{active_obj}` и `{active_group_name}` не могут использоваться совместно. Если ни один из них не указан, то транспортный агент создается как обычный пассивный агент.

Подробно теги `{channel_params}` и `{handshaking_params}` описываются ниже (см. 6.3 на стр. 43 и 6.4 на стр. 44).

Ошибка в конфигурационном файле или невозможность создания серверного сокета с указанным IP-адресом приводит к порождению фатальной ошибки (см. 4.3 на стр. 19).

Пример конфигурационного файла для создания серверного сокета:

```
{so_sysconf_ichannel
  {ip "0.0.0.0:15101"}

  {active_obj}
}
```

6.2 Исходящий канал

Созданием агентов для клиентских TCP/IP сокетов занимается библиотека `so_sysconf.ochannel.sysconf` (псевдоним `so_sysconf_2::ochannel`) и находящаяся в ней фабрика `so_sysconf_2::ochannel::factory`. Для ее использования в конфигурационный файл *so_sysconf* необходимо добавить инструкции:

```
|#
  Точка выхода из приложения по TCP/IP.
  Фабрика: so_sysconf_2::ochannel
#|
{load-dll "so_sysconf.ochannel.sysconf"
  {alias  "so_sysconf_2::ochannel" }
  {os-name-convert  "simple" }
}

|| Подключение к компоненту alice.
{make-coop
  {factory  "so_sysconf_2::ochannel::factory" }
  {coop "so_sysconf_2::ochannel::alice" }
  {cfg-file "ochannel.alice.cfg" }
}
|| Подключение к компоненту bob.
{make-coop
  {factory  "so_sysconf_2::ochannel::factory" }
  {coop "so_sysconf_2::ochannel::bob"}
  {cfg-file "ochannel.bob.cfg" }
}
```

Тег `{make-coop {coop}}` задает имя кооперации, которая будет создана фабрикой. Тег `{make-coop {cfg-file}}` задает имя конфигурационного файла с параметрами нового соединения. Для фабрики `so_sysconf_2::ochannel::factory` имя конфигурационного файла должно быть задано обязательно.

Важно: имя конфигурационного файла должно задаваться относительно пути к конфигурационным файлам из *app_paths* (см. 4.5 на стр. 21). Например, если при старте приложения в аргументе *–app-path-etc* было задано имя `../etc/manager/20060110`, то имя `ochannel.alice.cfg` будет преобразовано в `../etc/manager/20060110/ochannel.alice.cfg`.

Конфигурационный файл для фабрики `so_sysconf_2::ochannel::factory` имеет формат:

```
{so_sysconf_ochannel
  {addresses <str> [<str> [<str>...]] }
```

```
  [{reconnect_timeout <uint>}]
  [{restore_timeout <uint>}]
  [{try_switch_timeout <uint>}]
  [{channel_agent_name <str>}]
  {filter_agent_list <str> [<str> ...] }

  [{channel_params <параметры>}]
  [{handshaking_params <параметры>}]
}
```

Обязательными тегами являются `{addresses}` и `{filter_agent_list}`, остальные теги являются необязательными и могут отсутствовать.

Тег `{addresses}` задает один или несколько IP-адресов, к которым необходимо выполнять подключение транспортному агенту. Если задан только один IP-адрес, то ситуация проста: транспортный агент будет повторять попытки подключения к этому адресу до тех пор, пока соединение не будет установлено. А в случае разрыва соединения будет пытаться переподключиться к этому адресу.

Если же в `{addresses}` перечислено несколько IP-адресов, то транспортный агент использует возможности *so_alt_channel* (см. V на стр. 74). Первый из указанных адресов считается основным, а остальные — резервные, причем приоритет адреса убывает по мере приближения к концу списка (поэтому самый последний из IP-адресов является наименее приоритетным). В случае наличия нескольких IP-адресов транспортный агент сначала пытается установить соединение по основному IP-адресу (первому в списке). Если это не удалось, то пытается установить соединение по следующему IP-адресу и т.д. по кругу. Если соединение установлено по резервному адресу, то при при его разрыве попытки восстановления соединения начинаются опять с самого приоритетного адреса.

Тег `{filter_agent_list}` содержит список имен агентов, которые будут включены в фильтр для транспортного агента (т.е. через этот коммуникационный канал будут ходить только сообщения перечисленных в данном теге агентов).

Тег `{reconnect_timeout}` задает величину тайм-аута в секундах при повторении попыток подключения к IP-адресу. Этот тайм-аут отсчитывается после неудачной попытки установления соединения (в случае наличия только одного IP-адреса). По умолчанию имеет значение пять секунд.

Тег `{restore_timout}` задает величину тайм-аута в секундах при обнаружении разрыва соединения. Он отсчитывается после обнаружения разрыва текущего соединения перед первой попыткой повторного подключения (в случае наличия только одного IP-адреса). По умолчанию имеет значение в одну секунду.

Тег `{try_switch_timeout}` задает период попыток восстановления подключения по основному IP-адресу в случае нескольких IP-адресов. Значение указывается в секундах. По умолчанию: минута.

Тег `{channel_agent_name}` задает имя транспортного агента. По умолчанию имя транспортного агента формируется фабрикой на основе имени кооперации. Но, если транспортный агент должен получить конкретное имя (например, для того, чтобы можно было подписываться на его сообщения), то это имя можно задать посредством данного тега.

Подробно теги `{channel_params}` и `{handshaking_params}` описываются ниже (см. 6.3 на стр. 43 и 6.4 на стр. 44).

Агенты, создаваемые фабрикой `so_sysconf_2::ochannel::factory` являются членами активной группы, именем которой является имя кооперации. Поэтому лучше всего использовать с данным типом фабрики диспетчер с активными группами. Для других диспетчеров агенты окажутся просто пассивными агентами.

Невозможность обработки конфигурационного файла (его отсутствие или наличие ошибок) и невозможность по каким-то причинам зарегистрировать кооперацию с транспортным агентом порождает фатальную ошибку (см. 4.3 на стр. 19).

Примеры конфигурационных файлов для создания клиентского сокета:

1. Подключение к узлу порту 7000 узла `some.host.com` со стандартными параметрами переподключения. Фильтр разрешает транспорт сообщений только агента `mbapi_3::a_mbapi`:

```
{so_sysconf_ochannel
  {addresses "some.host.com:7000" }

  {filter_agent_list "mbapi_3::a_mbapi" }
}
```

2. Аналогично предыдущему, но величина тайм-аута при переподключений увеличивается до двадцати секунд, а переподключение инициируется сразу после обнаружения разрыва соединения:

```
{so_sysconf_ochannel
  {addresses "some.host.com:7000" }

  {reconnect_timeout 20 }
  {restore_timeout 0 }

  {filter_agent_list "mbapi_3::a_mbapi" }
}
```

3. Аналогично предыдущему, но с резервным каналом по адресу `mirror.host.com:8070`:

```
{so_sysconf_ochannel
  {addresses
    "some.host.com:7000"   || Основной адрес.
    "mirror.host.com:8070" || Резервный адрес.
  }

  {reconnect_timeout 20 }
  {restore_timeout 0 }

  {filter_agent_list "mbapi_3::a_mbapi" }
}
```

6.3 Управление параметрами канала

Параметры канала (как серверного, так и клиетского) задаются с помощью тега `{channel_params}` следующего вида:

```
{channel_params
  [{output_portion_size <uint>}]
  [{max_awaiting_buffer_size <uint>}]
  [{max_output_buffer_size <uint>}]
  [{input_portion_size <uint>}]
  [{in_threshold {package_count <uint>} {traffic_bulk <uint>}}]
  [{time_checking_period <uint>}]
```

```
  [{max_input_block_timeout <uint>}]
  [{max_output_block_timeout <uint>}]
}
```

Тег {output_portion_size} задает максимальный размер одной порции исходящих данных, записываемых в канал с помощью системных вызовов ОС. Значение по умолчанию — 32Kb.

Тег {max_awaiting_buffer_size} задает максимальный размер буфера ожидающих преобразования исходящих данных. Превышение этого объема приводит к принудительному закрытию канала. Значение по умолчанию — 512Kb.

Тег {max_output_buffer_size} задает максимальный размер буфера преобразованных и ожидающих отсылки исходящих данных. Превышение этого объема приводит к принудительному закрытию канала. Значение по умолчанию — 512Kb.

Тег {input_portion_size} задает максимальный размер одной порции входящих данных, считываемых из канала с помощью системных вызовов ОС. Значение по умолчанию — 32Kb.

Тег {in_threshold} задает порог входящих данных (с помощью двух обязательных дочерних тегов {package_count} и {traffic_bulk}), превышение которого останавливает чтение данных из канала. Значение по умолчанию — 1000 пакетов или 100Kb.

Тег {time_checking_period} задает темп, в секундах, с которым SObjectizet контролирует жизнеспособность канала. Значение по умолчанию — 1 секунда.

Тег {max_input_block_timeout} задает максимальное время, в секундах, в течении которого для канала запрещено чтение (из-за превышения входного порога). Превышение этого времени приводит к закрытию канала. Значению по умолчанию — 30 секунд.

Тег {max_output_block_timeout} задает максимальное время, в секундах, в течении которого для канала невозможно выполнять операции записи (например, из-за слишком медленного вычитывания данных на удаленной стороне). Значение по умолчанию — 30 секунд.

6.4 Управление параметрами процедуры handshake

Параметры процедуры handshake (как серверного, так и клиетского каналов) задаются с помощью тега {handshaking_params} следующего вида:

```
{handshaking_params
  [{compression}]
}
```

Если тег {compression} задан, то канал будет пытаться использовать zip-ование трафика (но только при условии, что удаленная сторона так же использует zip-ование).

Часть III

Generic Monitoring Tools

Глава 7

Введение

7.1 Назначение gemont

Библиотека *gemont* предназначена для того, чтобы написанные на SObjectizer приложения могли распространять мониторинговую информацию. Под мониторинговой информацией здесь понимаются данные информационно-измерительного характера, на основании которых можно судить о работе SObjectizer-приложения. Например, это могут быть имена состояний агентов, значения счетчиков транзакций, показатели текущей скорости обмена данными по коммуникационным каналам и т.д.

Распространение подобной мониторинговой информации требует решения нескольких задач:

- генерация мониторинговой информации;
- передача мониторинговой информации заинтересованным сторонам;
- получение и обработка мониторинговой информации.

Библиотека *gemont* предназначена для решения первых двух и содержит простой инструмент для решения третьей задачи.

7.2 Принцип работы gemont

В данном разделе коротко описываются основные понятия библиотеки *gemont* и действия, которые выполняются в библиотеке при выполнении тех или иных действий.

7.2.1 Источники данных и работа с ними

Ключевым элементом *gemont* является понятие *источника данных*. Источник данных — это именованая сущность, которая содержит текущее значение. Изменение источника данных приводит к распространению нового значения.

В программе источники данных представляются в виде C++ объектов специальных типов. Например:

```
1 // Здесь объявлены основные типы источников данных gemont_1.
2 #include <gemont_1/h/pub.hpp>
3 ...
4 class a_my_agent_t : public so_4::rt::agent_t
```

```
5   {
6   private :
7     // Источник данных типа "unsigned int".
8     gemont_1::scalar_data_source_t< unsigned int > m_counter_ds;
9     ...
10  };
11
12 a_my_agent_t::a_my_agent_t( ... )
13  : ... /* Инициализация базовых типов и пр. */
14  /* Назначение параметров источнику данных. */
15  , m_counter_ds(
16      // Его имя, которое должно быть уникальным.
17      "a_my_agent::counter",
18      // Имя его типа.
19      "a_my_agent_t::counter_ds",
20      // Его начальное значение.
21      0 )
22  { ... }
```

Для того, чтобы *gemont* узнал об источнике данных, источник данных нужно стартовать. Осуществляется это посредством вызова метода *start()*.

```
1 void
2 a_my_agent_t::so_on_subscription()
3   {
4     // Стартовать источник данных нужно только при запущеном
5     // SObjectizer Run-Time.
6     m_counter_ds.start();
```

Изменение источника данных происходит посредством вызова метода *set()*:

```
1 void
2 a_my_agent_t::evt_some_action()
3   {
4     ... /* Какие-то действия, в результате которых выясняется,
5            что источник данных должен получить новое значение. */
6     m_counter_ds.set( some_updated_value );
7   }
```

Когда источник данных должен прекратить свое существование для *gemont*, он должен быть остановлен посредством метода *stop()*:

```
1 void
2 a_my_agent_t::so_on_deregistration()
3   {
4     ...
5     /* Останавливаем источник данных. Если это не сделать, то
6        Gemont будет считать, что источник данных по прежнему
7        существует, но не изменяется. */
8     m_counter_ds.stop();
9   }
```

Примечание. В подавляющем большинстве случаев при использовании источников данных не приходится ни стартовать, ни останавливать их в ручную — это производится автоматически, речь об этом будет идти ниже.

7.2.2 Что скрывается за операциями над источниками данных

Ядром библиотеки *gemont* являются два глобальных агента: закрытый (private), имя которого является техническими деталями реализации *gemont*, и открытый (public), который используется различными средсвами обработки мониторинговой информации.

Когда выполняется старт источника данных (вызов его метода *start()*) источник данных отсылает специальное сообщение закрытого глобального агента. Когда значение источника данных изменяется (вызовом его метода *set()*) новое значение источника данных и время его модификации отсылается еще одним специальным сообщением закрытого глобального агента. Когда источник данных останавливается (вызов его метода *stop()*) источник данных отсылает третье специальное сообщение закрытого глобального агента.

Как раз то, что все манипуляции над источником данных приводят к отсылке сообщений SObjectizer-агента, и объясняет обязательное требование: **манипуляции над источником данных должны выполняться только при запущенном SObjectizer Run-Time**.

7.2.2.1 Почему используются сообщения глобального агента?

Сообщения глобального агента используются потому, что API SObjectizer позволяет обращаться к функции *so_4::api::make_global_agent* произвольное количество раз и не генерирует ошибку, если такой глобальный агент уже существует. Это позволяет легко подключать *gemont* в свои проекты: достаточно просто использовать источники данных, а все скрывающаяся за ними инфораструктура будет создаваться автоматически.

7.2.2.2 Зачем два глобальных агента?

Два глобальных агента необходимы для того, чтобы снизить объем мониторинговой информации, передаваемой во внешние приложения через SOP.

Дело в том, что изменения источников данных приводят к отсылке сообщений закрытого глобального агента. Эти сообщения могут возникать с высокой частотой — до десятков тысяч раз в секунду. Если эти сообщения будут уходить напрямую в какой-нибудь SOP-канал, то данный канал быстро окажется полностью занятым мониторинговой информацией.

Чтобы этого не происходило, в библиотеку *gemont* входит специальный компонент, *ретранслятор* мониторинговой информации (см. 10 на стр. 67). Он получает и обрабатывает все сообщения закрытого глобального агента, и выдает ее наружу уже в виде сообщений открытого глобального агента. Но уже с гораздо меньшей скоростью — не более нескольких изменений каждого источника данных в секунду.

Кроме минимизации трафика мониторинговой информации ретранслятор выполняет и другие задачи. В частности, с его помощью решается проблема получения списка источников данных при подключении к использующему *gemont* приложению. Например, пусть есть *server-side* приложение, написанное с использованием SObjectizer и *gemont*. К нему подключается GUI-инструмент, способный визуализировать мониторинговую информацию. Этому GUI-инструменту нужно получить список имеющихся в server-side приложении источников данных. Для этого GUI-инструмент отправляет в SOP-канал сообщение-запрос, принадлежащее открытому глобальному агенту *gemont*. Это сообщение получает ретранслятор мониторинговой информации и в ответ отсылает список источников данных в виде сообщений открытого глобального агента.

7.3 Имена источников данных, типы значений и имена типов источников данных

7.3.1 Имена источников данных

Имена источников данных должны быть уникальными. Сам *gemont* и построенные на его основе инструменты различают источники данных только по именам. Поэтому, если в приложении окажется несколько источников данных с одинаковыми значениями, то *gemont* не сможет диагностировать эту ситуацию и будет считать их значения значениями одного и того же источника данных (даже если значения будут иметь разный тип).

Обычно источникам данных дают имена, префиксом которых является имя агента, владеющего этими источниками данных. Например, пусть есть класс агента, выполняющего шифрование пакетов данных. У него должен быть источник данных — количество обработанных за последнюю секунду пакетов. Поскольку таких агентов в приложении может быть несколько (например, по числу CPU или подключенных к компьютеру аппаратных устройств для шифрования), поэтому имена источников данных этих агентов должны различаться. Это легко достигается за счет использования имени самого агента в качестве префикса имени его источников данных:

```
class a_package_encryptor_t
  : public so_4::rt::agent_t
  {
    ...
    // Источник данных, показывающий количество зашифрованных
    // за последнюю секунду пакетов.
    gemont_1::scalar_data_source_t< unsigned int > m_last_sec_packages;
  };

a_package_encryptor_t::a_package_encryptor_t(
  // Собственное имя агента.
  const std::string & self_name,
  // Остальные параметы конструктора.
  ... )
  : /* Инициализация базовых типов и других атрибутов. */
  , m_last_sec_packages(
      // Имя источника данных строится из имени агента.
      self_name + "::last_sec_packages",
      // Имя типа источника данных.
      "a_package_encryptor_t::m_last_sec_packages",
      // Начальное значение.
      0 )
  {}
```

7.3.2 Типы значений источников данных

Библиотека *gemont* версий 1.* умеет обрабатывать источники данных, значения которых имеют либо тип *unsigned int*, либо тип *std::string*. Значения остальных типов можно представлять в виде источников данных только путем преобразования значений либо в *unsigned int*, либо *std::string*.

Данное ограничение носит исторический характер — в момент реализации первой версии *gemont* требовалось работать только со значениями этих типов. Возможно, следующее поколение *gemont* — семейство версий 2.* — сможет поддерживать более широкий диапазон типов значений.

7.3.3 Имена типов источников данных

Каждый источник данных в приложении должен принадлежать какому-то типу (называему *dataclass* в *gemont*-терминологии). Имена типов, в отличии от имен источников данных, не должны быть уникальными — несколько источников данных вполне могут иметь одинаковые имена типов.

Имя типа источника данных указывает характер информации, распространяемой источником. Например, счетчик количества зашифрованных за последнюю секунду пакетов. Или процент успешных транзакций за последний час. Или текущее состояние агента. И т.д. и т.п.

Имя типа источника данных предназначено для того, чтобы инструменты, собирающие и обрабатывающие мониторинговую информацию могли настраивать свою работу. На основе *gemont* могут быть разработаны универсальные GUI-инструменты для визуализации значений источников данных[1]. Эти инструменты используют имена типов источников данных для определения способов отображения значений источников данных и, при необходимости, дополнительного привлечения внимания пользователей к отображаемой информации (звуковая и цветовая индикация, рассылка различных видов уведомлений). Например, если от источника данных S приходит значение 0, а источник данных принадлежит типу «процент успешных транзакций за последний час», то значение 0 будет свидетельствовать о проблемах с приложением или его окружением. А для информирования службы техподдержки об этой ситуации может быть выдан звуковой сигнал и/или отослано письмо по электронной почте. Если же значение 0 получено для источника данных типа «уровень помех в канале», то это значение, наоборот, свидетельствует об отличных условиях работы приложения, что будет отбражено соответствующим цветом или картинкой.

Обычно имя типа источника данных составляют из имени типа агента, в котором используется источник данных, и имени атрибута, который выступает в качестве источника данных:

```
1  a_package_encryptor_t::a_package_encryptor_t(
2    // Собственное имя агента.
3    const std::string & self_name,
4    // Остальные параметы конструктора.
5    ... )
6    : /* Инициализация базовых типов и других атрибутов. */
7    , m_last_sec_packages(
8        // Имя источника данных строится из имени агента.
9        self_name + "::last_sec_packages",
10       // Имя типа источника данных строится из имени типа агента.
11       "a_package_encryptor_t::m_last_sec_packages",
12       // Начальное значение.
13       0 )
14   {}
```

[1]Об одном из таких инструментов, разработанных в компании Интервэйл, речь пойдет ниже — см. 11.2 на стр. 70.

Глава 8

Основные классы источников данных

В данной главе речь пойдет о нескольких наиболее часто употребляемых классах источников данных *gemont*. Это классы, которые одновременно являются как источниками данных, так и свойствами SObjectizer-агентов (т.е. наследуются от двух классов). Такими классами являются шаблонный класс *gemont_1::scalar_data_source_as_trait_t* и класс *gemont_1::agent_state_data_source_t*.

Использование этих классов удобно тем, что программист освобожден от необходимости ручного вызова методов *start()* и *stop()* для источников данных:

- вызов *start()* выполняется автоматически при инициализации свойств агента при регистрации агента в SObjectizer;
- вызов *stop()* выполняется автоматически при деинициализации свойств агента при дерегистрации агента.

Т.о. при использовании источников данных, которые одновременно являются и свойствами агента, программисту требуется:

- создать источник данных;
- добавить его в список свойств агента;
- по мере необходимости изменять значение источника данных посредством его метода *set()*.

В конце главы будет рассказано о шаблонном классе *gemont_1::scalar_data_source_t*, который лежит в основе классов *gemont_1::scalar_data_source_as_trait_t* и *gemont_1::agent_state_data_source_t*. При использовании этого класса программисту необходимо вручную вызывать методы *start()* и *stop()*, однако, в некоторых случаях без этого не обойтись.

8.1 Шаблон scalar_data_source_as_trait_t

Шаблонный класс *gemont_1::scalar_data_source_as_trait_t* предназначен для создания источников данных, которые хранят одно значение и существуют все время, пока владеющий ими агент зарегистрирован в SObjectizer.

В *gemont* версий 1.* определены специализации *gemont_1::scalar_data_source_as_trait_t* для типов *unsigned int* и *std::string*.

Обычный сценарий использования *gemont_1::scalar_data_source_as_trait_t* заключается в следующем:

- программист объявляет источник данных атрибутом своего класса агента;
- в конструкторе класса агента программист инициализирует источник данных и добавляет его в список свойств агента;
- в обработчиках событий своего агента программист изменяет значения источника данных.

Вот как это может выглядеть для реализации счетчика зашированных в течении последней секунды пакетов в агенте, предназначенном для шифрования данных:

```
// Декларация класса агента.
class a_package_encryptor_t : public so_4::rt::agent_t
  {
    ...
    // Источник данных, который будет показывать, сколько
    // пакетов было зашифровано за последнюю секунду.
    gemont_1::scalar_data_source_as_trait_t< unsigned int > m_last_sec_packages;

  public :
    // Таких агентов в приложении может быть несколько,
    // поэтому каждому из них в конструктор будет передаваться
    // уникальное имя.
    a_package_encryptor_t(
      const std::string & self_name,
      // Остальные параметры...
      ... )
      : /* Инициализация базового типа и других атрибутов. */
        ...
      , m_last_sec_packages( /* Инициализация нашего источника данных. */
          // Имя строится на основе имени агента.
          self_name + "::last_sec_packages",
          // В качестве имени типа полное имя атрибута.
          "a_package_encryptor_t::m_last_sec_packages",
          // Начальное значение.
          0 )
      {
        ...
        // Источник данных должен быть добавлен в список свойств агента.
        so_add_traits( m_last_sec_packages );
      }

    ...
    // Сообщение, которое инициирует шифрование очередного пакета.
    struct msg_encrypt_package { ... };

    // Периодическое сообщение, которое отсчитывает секунды.
    struct msg_next_second { ... };
    ...
    // Реакция на необходимость шифрования еще одного пакета.
    void evt_encrypt_package( const msg_encrypt_package & cmd )
```

```
41        {
42          ... /* Выполнение действий по шифрованию. */
43          // Теперь можно инкрементировать значение источника данных.
44          // Метод current() возвращает его текущее значение.
45          m_last_sec_packages.set( m_last_sec_packages.current() + 1 );
46        }
47      ...
48      // Начало очередной секунды. Счетчик должен быть сброшен.
49      void evt_next_second()
50        {
51          m_last_sec_packages.set( 0 );
52        }
53      ...
54    };
```

8.2 Класс agent_state_data_source_t

Класс *gemont_1::agent_state_data_source_t* предназначен для представления в виде источника данных имени текущего состояния агента, владеющего этим источником данных. Каждый раз, когда агент изменяет свое состояние, автоматически изменяется значение источника данных.

Обычный сценарий использования *gemont_1::agent_state_data_source_t* заключается в следующем:

- в конструкторе класса агента программист создает и инициализирует источник данных, после чего добавляет его в список свойств агента;
- все остальное (старт, изменение значения, остановка) происходит с источником данных автоматически.

Объект типа *gemont_1::agent_state_data_source_t* даже не обязательно объявлять атрибутом агента — достаточно динамически создать его в конструкторе и добавить в список свойств агента через *agent_t::so_add_destroyable_trait()*.

В качестве примера использования *gemont_1::agent_state_data_source_t* можно рассмотреть некоего агента, взаимодействующего с удаленным компьютером по какому-то протоколу, например, UCP[1]. Агент должен дождаться установления TCP/IP соединения с удаленной стороной, провести аутентификацию и если она была успешной, начать обмен данными. Этот агент может иметь несколько состояний:

```
SOL4_CLASS_START( a_remote_client_t )
  ...
  // Состояние ожидания подключения к удаленной стороне.
  SOL4_STATE_START( st_disconnected )
    ...
  SOL4_STATE_FINISH()

  // Состояние ожидания результата аутентификации.
  SOL4_STATE_START( st_authenticating )
    ...
  SOL4_STATE_FINISH()

  // Состояние обмена данными по установленному соединению.
```

[1] Universal Computer Protocol.

```
  SOL4_STATE_START( st_connected )
    ...
  SOL4_STATE_FINISH()
SOL4_CLASS_FINISH()
```

Источник данных для мониторинга состояния такого агента должен создаваться в конструкторе:

```
a_remote_client_t::a_remote_client_t(
  // Имя агента.
  const std::string & self_name,
  // Дополнительные параметры агента.
  ... )
  : ... /* Инициализация базового типа и атрибутов. */
  {
    ...
    // Создание источника данных для информирования о текущем
    // состоянии агента.
    so_add_destroyable_trait(
        new gemont_1::agent_state_data_source_t(
            // Имя источника данных просто совпадает с именем агента.
            self_name,
            // Имя типа источника данных использует имя C++ класса
            // агента в качестве основы.
            "a_remote_client_t::agent_state" ) );
    ...
  }
```

Этих простых действий достаточно для того, чтобы был создан источник данных, имеющий имя своего агента и автоматически принимающий значения *st_disconnected*, *st_authenticating* или *st_connected* в зависимости от текущего состояния агента.

8.3 Шаблон scalar_data_source_t

Шаблонный класс *gemont_1::scalar_data_source_t* предназначен для создания источников данных, хранящих всего одно значение. Данный класс является базовым классом для *gemont_1::scalar_data_source_as_trait_t* и *gemont_1::agent_state_data_source_t*.

В *gemont* версий 1.* определены специализации *gemont_1::scalar_data_source_t* для типов *unsigned int* и *std::string*.

Главное отличие класса *gemont_1::scalar_data_source_t* от *gemont_1::scalar_data_source_as_trait_t* заключается в том, что *gemont_1::scalar_data_source_as_trait_t* предназначен только для источников данных, которые существуют столько же, сколько и владеющий ими агент. Тогда как источник данных на основе класса *gemont_1::scalar_data_source_t* может существовать гораздо меньшее время. Основной сценарий использования *gemont_1::scalar_data_source_t* сейчас — это связывание каких-то источников данных с коротко живущими объектами. Например, к SObjectizer-приложению подключается клиент. SObjectizer-приложение заводит для этого клиента новую сессию и связывает с данной сессией несколько источников данных (количество входящих/исходящих пакетов, общий объем трафика, средний объем пакета, средняя скорость обмена пакетами), созданных на основе *gemont_1::scalar_data_source_t*. При отключении клиента все связанные с ним источники данных разрушаются.

При использовании *gemont_1::scalar_data_source_t* программисту самому нужно заботится о старте и останове источника данных. Стартовать источник данных на основе *gemont_1::scalar_data_source_t* можно двумя способами:

- через метод *start()*:

```
typedef gemont_1::scalar_data_source_t< unsigned int > traffic_monitor_t;
traffic_monitor_t * monitor = new traffic_monitor_t( ... );
... /* Где-то чуть позже возникает возможность стартовать его. */
monitor->start();
```

- через передачу в конструктор *gemont_1::scalar_data_source_t* функции-стартера:

```
// Источник данных можно запустить сразу после создания.
traffic_monitor_t * monitor =
    new traffic_monitor_t(
        "some_name",          /* Имя источника данных. */
        "some_dataclass",     /* Имя типа источника данных. */
        0,                    /* Начальное значение. */
        &gemont_1::auto_start /* Автоматический старт источника. */
    );
```

Остановить источник данных можно либо обратившись к методу *stop()*, либо же уничтожив объект источника данных — останов автоматически произойдет в деструкторе.

В качестве примера использования *gemont_1::scalar_data_source_t* можно рассмотреть, как может выглядеть мониторинговая информация для упомяных выше сессий, которые некое SObjectizer-приложение создает для подключающихся к нему клиентов. Сначала определяется структура с источниками данных:

```
struct session_monitors_t
  {
    // Количество входящих пакетов.
    gemont_1::scalar_data_source_t< unsigned int > m_in_pkgs;
    // Количество исходящих пакетов.
    gemont_1::scalar_data_source_t< unsigned int > m_out_pkgs;
    ...

    // Конструктор инициализирует источники данных, поэтому
    // ему передается имя сессии, на основе которой создаются
    // имена источников данных.
    // Все источники данных автоматически стартуют.
    session_monitors_t(
      const std::string & session_name )
      : m_in_pkgs(
          session_name + "::in_pkgs",
          "session_monitors_t::in_pkgs",
          0,
          &gemont_1::start )
      , m_out_pkgs(
          session_name + "::out_pkgs",
          "session_monitors_t::out_pkgs",
          0,
          &gemont_1::start )
      , ...
      {}
  };

// Умный указатель для session_monitors_t для того, чтобы можно
// было сохранять динамически-созданные session_monitors_t в
```

```
// стандартных C++ контейнерах.
// Примечание. Класс умного указателя из библиотеки ACE используется
// просто потому, что ACE лежит в основе SObjectizer и Gemont.
typedef ACE_Refcounted_Auto_Ptr<
        session_monitors_t,
        ACE_Null_Mutex >
    session_monitors_shared_ptr_t;

// Тип карты соответствия между именами сессий и мониторинговой
// информацией для сессии.
typedef std::map<
        std::string,
        session_monitors_shared_ptr_t >
    session_monitors_map_t;
```

Затем эти структуры используются в агенте, отвечающем за работу с клиентами:

```
class a_session_manager_t : public so_4::rt::agent_t
  {
    ...
    // Мониторинговая информация для активных сессий.
    session_monitors_map_t m_session_monitors;

  public :
    ...
    // Реакция на подключение нового клиента.
    void evt_client_connected(
      const so_4::rt::comm::msg_client_connected & cmd )
      {
        ...
        // Каким-то образом формируется имя сессии.
        const std::string session_name = ...;
        // Создается мониторинговая информация для сессии.
        session_monitors_shared_ptr_t monitors(
            new session_monitors_t( session_name ) );
        m_session_monitors[ session_name ] = monitors;
        // Теперь мониторинговая информация создана и все
        // новые источники данных запущены.
        ...
      }
    // Реакция на отключение клиента.
    void evt_client_disconnected(
      const so_4::rt::comm::msg_client_disconnected & cmd )
      {
        ...
        // Каким-то образом формируется имя сессии.
        const std::string session_name = ...;
        // Информация о сессии уничтожается.
        m_session_monitors.erase( session_name );
        // Теперь все созданные ранее источники данных
        // остановлены и уничтожены.
        ...
      }
  };
```

Глава 9

Дополнительные классы

В данной главе рассматриваются классы *gemont_1::value_holder_as_trait_t* (*gemont_1::value_holder_t*) и *gemont_1::temporary_sources_t*. Они не являются основными классами библиотеки *gemont* — без них вполне можно обойтись. Но временами они существенно облегчают использование *gemont* источников данных.

9.1 Классы value_holder_as_trait_t и value_holder_t

9.1.1 Назначение и использование

Основная сложность с классами *gemont_1::scalar_data_source_t* в том, что для них нужно явно вызывать метод *set()* для того, чтобы выполнить обновление источника данных.

Например, в простом случае есть счетчик активных транзакций. Этот счетчик используется агентом-владельцем для проверки условия: если количество транзакций больше некой величины, то новые транзакции к обработке не принимаются. Программист может оформить счетчик транзакций и источник данных для него двумя способами:

- во-первых, можно сделать счетчик и источник данных разными атрибутами агента:

```
class a_trx_manager_t : public so_4::rt::agent_t
  {
    // Вот сам счетчик.
    unsigned int m_trx_count;
    // А вот и источник данных для него.
    gemont_1::scalar_data_source_as_trait_t< unsigned int > m_trx_count_ds;
    ...
  };
...
void a_trx_manager_t::evt_new_trx( const msg_new_trx & cmd )
  {
    // Вот используется счетчик.
    if( m_trx_count < max_parallel_transactions_count )
      {
        ...
        // И еще раз используется.
        ++m_trx_count;
        // А здесь уже обновляется его источник данных.
        m_trx_count_ds.set( m_trx_count );
```

```
20        }
21      ...
22    }
```

Данный вариант плох тем, что со временем может произойти рассинхронизация операций над счетчиком и над источником данных, в результате чего источник данных будет показывать неправильные значения;

- во-вторых, можно сделать счетчиком сам источник данных:

```
1  class a_trx_manager_t : public so_4::rt::agent_t
2    {
3      // Счетчик и источник данных в одном лице.
4      gemont_1::scalar_data_source_as_trait_t< unsigned int > m_trx_count;
5      ...
6    };
7  ...
8  void a_trx_manager_t::evt_new_trx( const msg_new_trx & cmd )
9    {
10     // Вот используется счетчик.
11     if( m_trx_count.current() < max_parallel_transactions_count )
12       {
13         ...
14         // И еще раз используется. И сразу же обновляется.
15         m_trx_count.set( m_trx_count.current() + 1 );
16       }
17     ...
18   }
```

Вроде бы уже лучше — рассинхронизации произойти не может. Но появляется другая проблема — писать код, в котором для работы с целочисленным счетчиком нужно вызвать методы *current()* и *set()*, т.е. писать больше кода, чем раньше. Что так же может привести к ошибкам, глупым и трудноуловимым: `m_trx_count.set(m_trx_count.current())` вместо `m_trx_counte.set(m_trx_count.current()+1)`.

Могут встречаться и более сложные случаи когда нет отдельного счетчика транзакций, а есть вектор с описаниями текущих транзакций и источник данных должен отображать количество элементов в векторе. При использовании *gemont_1::scalar_data_source_t* у разработчика нет другого выбора, кроме разделения атрибутов — есть отдельно вектор транзакций и есть источник данных. Из-за чего возникает опасность забыть обновить источник данных при обновлении вектора транзакций. А хотелось бы, чтобы после любой операции изменения вектора источник данных автоматически изменялся... Как раз для этого предназначены классы *gemont_1::value_holder_t*!

Идея *gemont_1::value_holder_t* в том, что *gemont_1::value_holder_t* содержит как источник данных, так и нужный программисту объект (счетчик транзакций или вектор с описаниями транзакций) и предоставляет доступ к объекту через переопределенный *operator**. Причем сам *gemont_1::value_holder_t* проверяет, изменился ли объект после обращения к нему. И если изменился, то сам обновляет источник данных. Выглядит это следуюшим образом.

- для случая целочисленного счетчика:

```
1  class a_trx_manager_t : public so_4::rt::agent_t
2    {
3      // Счетчик и источник данных в одном лице.
```

```
    gemont_1::value_holder_as_trait_t< unsigned int > m_trx_count;
    ...
  };
...
void a_trx_manager_t::evt_new_trx( const msg_new_trx & cmd )
  {
    // Вот используется счетчик.
    if( **m_trx_count < max_parallel_transactions_count )
      {
        ...
        // И еще раз используется. И сразу же обновляется.
        ++(**m_trx_count);
      }
    ...
  }
```

- для случая вектора описаний транзакций:

```
class a_trx_manager_t : public so_4::rt::agent_t
  {
    // Тип вектора описаний транзакций.
    typedef std::vector< trx_description_t > trx_container_t;

    // Вектор транзакций и источник данных в одном лице.
    gemont_1::value_holder_as_trait_t<
            // Тип объекта, который хранит value_holder.
            trx_container_t,
            // Тип значения у источника данных.
            // Нужно помнить, что gemont_1 может работать только
            // с unsigned int и std::string.
            // Здесь нужен unsigned int, т.к. источник данных
            // отображает размер вектора.
            unsigned int,
            // А вот эта конструкция как раз и отвечает за то,
            // чтобы источник данных показывал размер вектора.
            gemont_1::stl_container_size< trx_container_t > >
        m_transactions;
    ...
  };
...
void a_trx_manager_t::evt_new_trx( const msg_new_trx & cmd )
  {
    // Вот используется вектор транзакций.
    if( (**m_trx_count).size() < max_parallel_transactions_count )
      {
        ...
        // И еще раз используется. И сразу же обновляется.
        (**m_trx_count).push_back( new_trx_description );
      }
    ...
  }
```

В остальном использование классов *gemont_1::value_holder_t* похоже на использование классов *gemont_1::scalar_data_source_t*:

- при использовании *gemont_1::value_holder_as_trait_t* программист должен:
  - объявить объект *gemont_1::value_holder_as_trait_t* атрибутом агента;
  - в конструкторе агента объект *gemont_1::value_holder_as_trait_t* должен быть проинициализирован и добавлен в список свойств агента;
- при использовании *gemont_1::value_holder_t* программист должен:
  - создать и проинициализировать объект *gemont_1::value_holder_t*;
  - стартовать объект *gemont_1::value_holder_t* посредством его метода *start()*. При необходимости старта сразу по создании объекта, можно передать в конструктор *gemont_1::value_holder_t* указатель на функцию-стартер (точно так же, как это происходит для класса *gemont_1::scalar_data_source_t*);
  - остановить объект *gemont_1::value_holder_t* когда он становится не нужным (посредством метода *stop()*).

9.1.2 Откуда ноги растут

Идея *gemont_1::value_holder_t* была позаимствована из работы Б.Страуструпа «Wrapping C++ Member Function Calls» [WRAPM]. Суть в том, что переопределенный *operator** возвращает временный объект-*proxy*, для которого так же переопределен *operator**. Именно *operator** proxy-объекта возвращает ссылку на содержащееся в *gemont_1::value_holder_t* значение.

Деструктор proxy-объекта (который вызывается после завершения всего выражения, в котором он был создан) проверяет, изменилось ли значение хранящегося в *gemont_1::value_holder_t* объекта. Если изменилось, то происходит обновление источника данных.

При использовании *gemont_1::value_holder_t* полезно понимать, что в *gemont_1::value_holder_t* есть два варианта *operator**:

- неконстантный, который возвращает неконстантный proxy-объект. А деструктор неконстантного proxy-объекта уже выполняет проверки и обновление источника данных;
- константный, который возвращает константный proxy-объект. В свою очередь *operator** константного proxy возвращает константную ссылку на хранящийся в *gemont_1::value_holder_t* объект. А деструктор константного proxy ничего не делает, т.к. предполагается, что хранящийся в *gemont_1::value_holder_t* объект по константной ссылке не может быть изменен.

То, что контантный *operator** в *gemont_1::value_holder_t* не приводит к обновлению источника данных положительным образом сказывается на скорости работы. Поэтому рекомендуется работать с *gemont_1::value_holder_t* через константные ссылки/указатели.

Так же существует теоритическая возможность, что *gemont_1::value_holder_t* будет владеть объектом с помеченными модификатором *mutable* атрибутами. В этом случае обращение к объекту через константный *operator** сможет изменить объект, но *gemont_1::value_holder_t* не отразит эти изменения в источнике данных, т.к. константный proxy не обновляет источник данных.

9.1.3 Более сложные сценарии использования

Шаблон *gemont_1::value_holder_t* имеет четыре параметра:

```
template<
  //! Тип данных, ссылка на который будет возвращаться operator*().
  class ACCESS_TYPE,
  //! Тип данных, который хранится в data-source.
  class DATA_SOURCE_VALUE_TYPE = ACCESS_TYPE,
  //! Тип извлекателя значения из Access_type для помещения в data-source.
  class VALUE_EXTRACTOR = simple_get< DATA_SOURCE_VALUE_TYPE, ACCESS_TYPE >,
  //! Тип data-source.
  class DATA_SOURCE = scalar_data_source_t< DATA_SOURCE_VALUE_TYPE > >
class value_holder_t
  {
  protected :
    //! Значение, к которому нужно обращаться через operator*().
    ACCESS_TYPE m_v;
    //! Источник данных, в котором отражаются изменения m_v.
    DATA_SOURCE m_ds;
...
```

Параметр *ACCESS_TYPE* указывает, объект какого типа будет храниться в *gemont_1::value_holder_t*. Ссылка на этот тип будет возвращаться в результате работы всех переопределенных методов *operator**.

Параметр *DATA_SOURCE_VALUE_TYPE* указывает, какое значение будет использоваться источником данных. По умолчанию типы *ACCESS_TYPE* и *DATA_SOURCE_VALUE_TYPE* совпадают. Но во многих случаях они оказываются разными: *ACCESS_TYPE* является типом некоторого контейнера, а *DATA_SOURCE_VALUE_TYPE* либо *unsigned int* для мониторинга размера контейнера, либо *std::string* для какого-то описания содержимого контейнера.

Параметр *VALUE_EXTRACTOR* — это тип функтора, который преобразует значение *ACCESS_TYPE* в значение *DATA_SOURCE_VALUE_TYPE*. По умолчанию используется штатный функтор *simple_get*, который возвращает исходное значение в качестве результирующего. Этот функтор не подойдет, если *ACCESS_TYPE* и *DATA_SOURCE_VALUE_TYPE* являются несовместимыми друг с другом типами (например, *std::vector* и *unsigned int*). Для таких случаев в *gemont* есть еще один штатный функтор — *stl_container_size*, который считает, что ему передают в качестве параметра STL-контейнер, а он возвращает размер этого контейнера. В остальных случаях программисту потребуется реализовать свой функтор.

Параметр *DATA_SOURCE* определяет тип источника данных. По умолчанию используется *gemont_1::scalar_data_source_t*.

В качестве примера довольно сложного сценария использования класса *gemont_1::value_holder_as_trait_t* здесь приводится схематическая иллюстрация одного решения, которое было использовано в реальном проекте. Одному из агентов этого проекта требовалось хранить карту клиентов, транзакции которых агент обслуживает в данный момент. Для каждого клиента имелся список обрабатываемых транзакций, список отложенных до наступления определенного события транзакций и список ожидающих своей очереди транзакций, которые еще не обрабатывались. Необходимо было выдавать краткую справку о содержимом этой карты клиентов в виде мониторинговой информации.

Решение состояло в том, чтобы поместить карту клиентов в *gemont_1::value_holder_as_trait_t* и определить параметр *VALUE_EXTRACTOR* таким образом, чтобы содержимое карты клиентов преобразовывалось в строку. Вот как это приблизительно выглядело:

```
// Хранилище информации о транзакциях одного клиента.
class client_trx_info_t
  {
    ...
    unsigned int trx_in_progress_count() const;
```

```
6     unsigned int delayed_trx_count() const;
7     unsigned int awaiting_trx_count() const;
8   };
9  // Умный указатель для client_trx_info_t.
10 typedef ACE_Refcounted_Auto_Ptr< ... > client_trx_info_ptr_t;
11
12 // Тип карты клиентов.
13 // client_id_t -- это тип идентификатора клиента (кроме прочего
14 // для него определен оператор сдвига в std::ostream).
15 typedef std::map<
16         client_id_t,
17         client_trx_info_ptr_t >
18     client_map_t;
19
20 // Функтор, который делает краткую справку по содержимому client_map_t.
21 struct client_map_brief_describer_t
22   {
23     // Формирование краткой справки.
24     std::string
25     operator()( const client_map_t & clients ) const
26       {
27         // Для каждого клиента строится описание в отдельной строке.
28         std::ostringstream s;
29         for( client_map_t::const_interator
30             it = clients.begin(), it_end = clients.end();
31             it != it_end;
32             ++it )
33           s << it->first
34               << ": in_progress: " << it->second->trx_in_process_count()
35               << ", delayed: " << it->second->delayed_trx_count()
36               << ", awaiting: " << it->second->awaiting_trx_count()
37               << std::endl;
38
39         return s.str();
40       }
41   };
42
43 // Тип value_holder-а для client_map-а.
44 typedef gemont_1::value_holder_as_trait_t<
45         client_map_t,
46         std::string,
47         client_map_brief_describer_t >
48     client_map_holder_t;
49
50 // Затем, где-то в агенте-обработчике...
51 class a_client_trx_processor_t : public so_4::rt::agent_t
52   {
53     ...
54     // ... объявляется карта транзакций клиентов...
55     client_map_holder_t m_client_map;
56     ...
57     void try_start_new_trx(
58         const msg_new_trx & cmd,
59         client_map_t & actual_map ) { ... }
```

```
60      ...
61      // ...которая затем используется.
62      void evt_new_trx( const msg_new_trx & cmd )
63        {
64          try_start_new_trx( **m_client_map ); /* И сразу после ее
65                                                  использования автоматически
66                                                  обновляется источник
67                                                  данных. */
68          ...
69        }
70      ...
71    };
```

9.1.4 Осторожность не помешает

На первый взгляд, использование *gemont_1::value_holder_t* выглядит более удобным, чем *gemont_1::scalar_data_source_t*. Поэтому *gemont_1::value_holder_t* начинает использоваться везде без оглядки на то, к каким накладным расходам может привести его использование. Что обязательно скажется, когда количество модификаций источника данных превысит несколько тысяч раз в секунду.

Дело в том, что обращение к *gemont_1::value_holder_t* более накладно, чем к *gemont_1::scalar_data_source_t* за счет создания и разрушения вспомогательных proxy-объектов, а так же за счет преобразований значений из одного типа в другой. И если единственное обращение к *gemont_1::value_holder_t* в обработчике одного события еще не страшно, то вот два-три (иногда и больше) обращений уже могут создавать совершенно лишнюю нагрузку. Вот, например, как это может происходить (см. пример из предыдущего раздела):

```
1  // Реализация реакции на начало новой транзакции без учета
2  // накладных расходов value_holder-ов.
3  void evt_new_trx( const msg_new_trx & cmd )
4    {
5      // В случае отсутствия ресурсов для новой транзакции нужно
6      // попробовать удалить просроченные транзакции.
7      if( !is_free_slot_for_trx_exists( cmd, **m_client_map ) )
8        {
9          try_remove_too_old_trx( **m_client_map );
10         try_initiate_long_awaiting_trx( **m_client_map );
11       }
12
13     // Теперь уже можно пытаться начинать новую транзакцию.
14     try_star_new_trx( cmd, **m_client_map );
15   }
```

Эта простая реализация *evt_new_trx* приводит к четырем преобразованиям значения *m_client_map*. Хотя можно было бы обойтись всего одним, если бы программист вынес код *evt_new_trx* в отдельный метод:

```
1  // Реализация реакции на начало новой транзакции без учета
2  // накладных расходов value_holder-ов.
3  void evt_new_trx( const msg_new_trx & cmd )
4    {
5      evt_new_trx_i( cmd, **m_client_map );
6    }
7
```

```
// Фактическая реализация реакции на новую транзакцию.
void evt_new_trx_i(
  const msg_new_trx & cmd,
  client_map_t & client_map )
  {
    // В случае отсутствия ресурсов для новой транзакции нужно
    // попробовать удалить просроченные транзакции.
    if( !is_free_slot_for_trx_exists( cmd, client_map ) )
      {
        try_remove_too_old_trx( client_map );
        try_initiate_long_awaiting_trx( client_map );
      }

    // Теперь уже можно пытаться начинать новую транзакцию.
    try_star_new_trx( cmd, client_map );
  }
```

Поэтому в большинстве случаев надежнее пользоваться *gemont_1::scalar_data_source_t*. Но и слишком частое обращение к *gemont_1::scalar_data_source_t* так же способно создать дополнительную, а иногда и совершенно лишнюю, нагрузку на использующее *gemont* SObjectizer-приложение. Очень часто не требуется обновлять мониторинговую информацию непосредственно при изменении каких-либо значений, а достаточно темпа обновления от пяти до двух раз в секунду, а иногда и один раз в несколько секунд. В таких случаях гораздо проще и дешевле (как в плане накладных расходов, так и удобства реализации) завести периодическое сообщение с нужным темпом и обновлять источники данных агента в обработчике этого сообщения. Т.е. обсуждавшийся выше пример с картой клиентов мог бы быть реализован так:

```
// Создание краткого описания карты клиентов.
std::string
make_brief_client_map_description( const client_map_t & clients ) const
  {
    ...
  }
// Затем, где-то в агенте-обработчике...
class a_client_trx_processor_t : public so_4::rt::agent_t
  {
    ...
    // ... объявляется карта транзакций клиентов...
    client_map_t m_client_map;
    // ... и источник данных для нее...
    gemont_1::scalar_data_source_as_trait_t< std::string > m_client_map_ds;
    ...
    void try_start_new_trx(
        const msg_new_trx & cmd,
        client_map_t & actual_map ) { ... }
    ...
    // Карта клиентов просто используется.
    void evt_new_trx( const msg_new_trx & cmd )
      {
        if( !is_free_slot_for_trx_exists( cmd, m_client_map ) )
          ...
      }
    // А источник данных просто обновляется.
    void evt_update_gemont_data()
```

```
28        {
29          m_client_map_ds.set(
30              make_brief_client_map_description( m_client_map ) );
31          ...
32        }
33      ...
34    };
```

9.2 Класс temporary_sources_t

Класс *gemont_1::temporary_sources_t* предназначен для упрощения работы с источниками данных, создаваемых на непродолжительное время. Фактически, *gemont_1::temporary_sources_t* представляет из себя карту, где ключем является некий идентификатор источника данных (по умолчанию таким идентификатором является строка, которая так же является именем источника данных), а значением — динамически созданный объект типа *gemont_1::scalar_data_source_as_trait_t*.

Класс *gemont_1::temporary_sources_t* берет на себя задачу автоматического запуска источника данных при добавлении в хранилище, автоматического останова источника данных (либо при изъятии из хранилища, либо при дерегистрации агента, владеющего *gemont_1::temporary_sources_t*), автоматического уничтожения источников данных (как при изъятии, так и при дерегистрации агента-владельца).

Класс *gemont_1::temporary_sources_t* является свойством агента, поэтому работа с *gemont_1::temporary_sources_t* во многом напоминает работу с *gemont_1::scalar_data_source_as_trait_t* и *gemont_1::value_holder_as_trait_t*:

- объект *gemont_1::temporary_sources_t* должен быть объявлен атрибутом класса агента, который его использует;
- объект *gemont_1::temporary_sources_t* должен быть добавлен в список свойств агента.

В качестве примера использования *gemont_1::temporary_sources_t* можно рассмотреть приложение, к которому подключаются клиенты. Для каждого клиента нужно иметь источник данных, который содержит время получения последнего запроса клиента. Выглядеть это может следующим образом:

```
1  // Агент, обслуживающий клиентов.
2  class a_client_manager_t : public so_4::rt::agent_t
3    {
4      // Тип источника данных для одного клиента.
5      // Поскольку в gemont_1 нет "родной" поддержки даты и времени,
6      // то дата и время представляется в виде строки.
7      typedef gemont_1::scalar_data_source_as_trait_t< std::string >
8          last_activity_timestamp_monitor_t;
9
10     // Тип хранилища даты последней активности для клиентов.
11     typedef gemont_1::temporary_sources_t<
12             last_activity_timestamp_monitor_t >
13         last_activity_timestamps_t;
14
15     // Само хранилище дат последней активности.
16     last_activity_timestamps_t m_activity_timestamps;
17     ...
```

```
18  public :
19    // В конструкторе нужно добавить activity_timestamps в
20    // список свойств агента.
21    a_client_manager_t( ... )
22      {
23        ...
24        so_add_traits( m_activity_timestamps );
25      }
26    ...
27    // Реакция на запрос клиента.
28    void evt_client_request(
29      const so_4::rt::event_data_t & event_data,
30      const msg_request & cmd )
31      {
32        ...
33        // Если это первый запрос клиента, то нужно создавать
34        // источник данных.
35        const std::string monitor_name = create_monitor_name(
36            event_data.channel() );
37        const std::string current_timestamp = create_timestamp();
38
39        last_activity_timestamp_monitor_t * monitor =
40            m_activity_timestamps.find( monitor_name );
41        if( !monitor )
42          {
43            monitor = new last_activity_timestamp_monitor_t(
44                monitor_name,
45                "a_client_manager_t::last_activity_timestamp",
46                current_timestamp );
47            m_activity_timestamps.insert( monitor_name, monitor );
48          }
49        else
50          monitor->set( current_timestamp );
51
52      }
53    ...
54    // Реакция на отключение клиента.
55    void evt_client_disconnected(
56      const so_4::rt::comm::msg_client_disconnected & cmd )
57      {
58        const std::string monitor_name = create_monitor_name(
59            cmd.m_channel );
60        m_activity_timestamps.erase( monitor_name );
61      }
62  };
```

Глава 10

Ретранслятор

10.1 Назначение

Первой задачей *gemont*-ретранслятора является выдача мониторинговой информации из SObjectizer-приложения наружу с через коммуникационные SOP-каналы. Источники данных могут отсылать сообщения об изменении своих значений с очень высоким темпом (до нескольких десятков тысяч сообщений в секунду). Если они все будут перенаправляются в коммуникационные каналы, то коммуникационные каналы могут просто не справится с нагрузкой. Поэтому *gemont* использует сообщения двух глобальных агентов — открытого и закрытого. Сообщения закрытого глобального агента не предназначены для передачи во внешний мир. Зато сообщения открытого глобального агента используются именно для этого.

Агент-ретранслятор в *gemont* получает сообщения закрытого глобального агента и собирает значения источников данных. Несколько раз в секунду он ретранслирует собранные значения в виде сообщений открытого глобального агента. Если какой-то источник данных обновляется слишком часто, то его значения ретранслируются не более трех раз в секунду.

За счет такого режима работы агента-ретранслятора резко сокращается объем мониторинговой информации, распространяемой по коммуникационным каналам.

Второй задачей *gemont*-ретранслятора является выдача внешнему приложению списка имеющихся источников данных и их последних значений. Для этого внешнее приложение должно подключиться к контролируемому SObjectizer-приложению с помощью коммуникационного SOP-канала и отослать в канал сообщение *msg_get_data_source_info*. В ответ на него *gemont*-ретранслятор отошлет в этот канал описания известных ему источников данных и их последние значения.

В библиотеке *gemont* ретранслятор оформлен в виде отдельной DLL (проектный файл `gemont_1/retranslator/prj.rb`) и его нужно специальным образом запускать.

10.2 Ручной запуск ретранслятора

Для запуска *gemont*-ретранслятора вручную необходимо:

- зарегистрировать в какой-либо кооперации приложения агента *gemont_1::retranslator::a_retranslator_t* (он описан в заголовочном файле `gemont_1/retranslator/h/pub.hpp`);
- добавить в список подпроектов проектный файл `gemont_1/retranslator/prj.rb`.

10.3 Запуск через so_sysconf

При использовании подсистемы *so_sysconf* (см. II на стр. 11) запустить *gemont*-ретранслятор можно добавив в sysconf-скрипт приложения следующий фрагмент:

```
1 |#
2   Средства ретрансляции мониторинговой информации.
3   Содержат кооперацию: gemont_1::retranslator
4 #|
5 {load-dll  "gemont.retranslator.sysconf"
6   {os-name-convert  "simple"}
7   {alias  "gemont_1::retranslator::sysconf"}
8 }
9 {reg-coop  "gemont_1::retranslator" }
```

Так же необходимо скомпилировать DLL-библиотеку `gemont.retranslator.sysconf` при помощи проектного файла `gemont_1/retranslator/sysconf/prj.rb`.

Глава 11

Средства визуализации мониторинговой информации

11.1 Сброс мониторинговой информации в файл

В состав *gemont* входит небольшой дополнительный инструмент *gemont-snapshot*, который отвечает за периодический сброс текущей мониторинговой информации в текстовый файл. Для использования *gemont-snapshot* его нужно подключить к SObjectizer-приложению (это отдельная библиотека) и после своего старта *gemont-snapshot* будет с заданным темпом создавать «снимки» текущего состояния источников данных и записывать их в указанный текстовый файл.

На данный момент *gemont-snapshot* состоит из одного агента, который получает сообщения закрытого глобального агента и собирает значения источников данных. При наступлении времени формирования очередного «снимка» он записывает все имеющиеся у него значения в файл.

11.1.1 Ручной запуск gemont-snapshot

Для запуска *gemont-snapshot* необходимо:

- зарегистрировать в какой-либо кооперации приложения агента *gemont_1::snapshot::a_snapshot_t* (он описан в заголовочном файле `gemont_1/snapshot/h/pub.hpp`);
- добавить в список подпроектов проектный файл `gemont_1/snapshot/prj.rb`.

11.1.2 Запуск через so_sysconf

При использовании подсистемы *so_sysconf* (см. II на стр. 11) запустить *gemont-snapshot* можно добавив в sysconf-скрипт приложения следующий фрагмент:

```
1 |#
2   Средства мониторинга текущего состояния системы.
3 #|
4 {load-dll  "gemont.snapshot.sysconf"
5   {os-name-convert  "simple"}
6   {alias  "gemont_1::snapshot::sysconf"}
7 }
8 {make-coop
```

```
  {factory  "gemont_1::snapshot::sysconf" }
  {coop  "gemont_1::snapshot::local" }
  {cfg-file  "etc/gemont_1/snapshot/local.cfg" }
}
```

Где имя `etc/gemont_1/snapshot/local.cfg` является именем конфигурационного файла с параметрами для *gemont-snapshot*. Этот файл имеет формат:

```
{snapshot
  {.file_name <str> }
  {.date_time_format <str> }
}
```

Тег `.file_name` задает имя файла, в который будет помещаться очередной «снимок». Тег `.date_time_format` определяет формат записи даты и времени изменения источника данных. Для задания формата даты/времени используется формат C-шной функции *strftime*.

Пример конфигурационного файла для *gemont-snapshot*:

```
{snapshot
  || Имя файла для "снимков".
  {.file_name  "log/snapshot.log" }

  || Формат отображения даты/времени.
  {.date_time_format  "%Y.%m.%d %H:%M:%S" }
}
```

Так же необходимо скомпилировать DLL-библиотеку `gemont.snapshot.sysconf` при помощи проектного файла `gemont_1/snapshot/sysconf/prj.rb`.

11.2 Globe

В компании Интервэйл разработан специальный GUI-инструмент для мониторинга SObjectizer-приложений — Globe. Globe подключается к SObjectizer-приложениям с помощью SOP-каналов, собирает и визуализирует получаемую от SObjectizer-приложений мониторинговую информацию. При визуализации используются правила, определяемые пользователями Globe. Эти правила строятся на основе типов источников данных и могут содержать различные реакции (цветовые и звуковые) на изменения значений источников данных этих типов, например:

```
{dataclass aag_3::workaround::send::result_dist::percentage::success
  {type uint}
  {priority 3}

  {if {< 20}
    {level error}
    {log "Очень много ошибок при отправке SMS"}
  }
  {if {< 70}
    {level warn}
  }
  {otherwise
    {level info}
  }
}
```

Globe распространяется компанией Интервэйл в виде отдельного продукта.

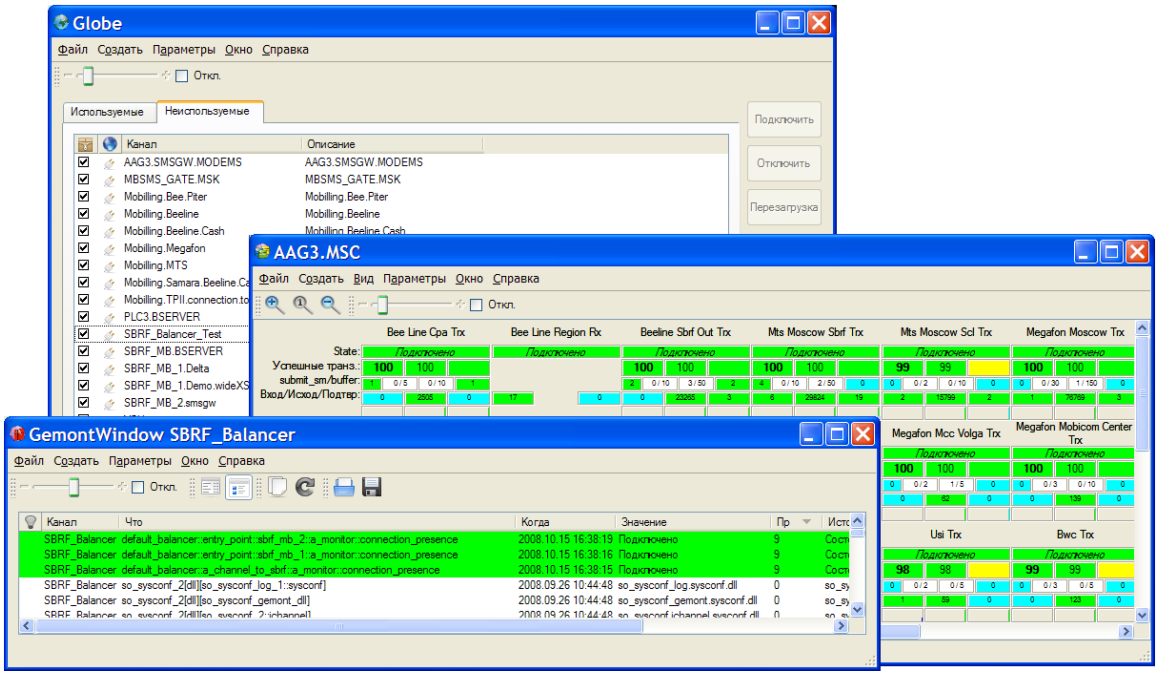

Рис. 11.1: Снимок экрана во время работы приложения Globe.

Часть IV

SObjectizer Logger

Часть V

SObjectizer Alternative Channel

Глава 12

Поддержка альтернативных каналов в so_alt_channel

12.1 Описание проблемы

Идея каналов ввода-вывода в SObjectizer базируется на том, что каждый канал предназначен для соединения с одним конкретным узлом. Поэтому клиентские транспортные агенты в SObjectizer (*so_4::rt::comm::a_sop_outgoing_channel_t*, *so_4::rt::comm::a_raw_outgoing_channel_t*) работают с одним каналом, который держит подключение к одному узлу. При этом, если соединение рвется, то осуществляются попытки восстановить соединение только с этим узлом.

Но на практике возникают задачи, в которых при разрыве основного соединения нужно перейти на резервный канал. В этом случае использование транспортных агентов SObjectizer вызывает сложности. Можно, например, создавать нескольких агентов, по одному на каждое соединение, и управлять ими. Но в этом случае нужно не только отслеживать сообщения `msg_client_connected`, `msg_client_disconnected` от нескольких агентов, но еще и контролировать, какой агент потерял соединение, и какого агента нужно заставить установить новое соединение.

Более удобен вариант создания специального класса, реализующего интерфейс *so_4::transport_layer::connector_controller_t*. Такой класс мог бы получать в конструкторе список реальных connector_controller-ов, по одному на каждый канал. В методе *connect()* этот класс последовательно перебирал бы все фабрики, пока не установил бы соединение.

Проект *so_alt_channel* как раз предоставляет описанную реализацию интерфейса *so_alt_channel_2::cyclic_connector_controller_t* и набор средств для преодоления двух вышеуказанных проблем.

12.2 Основная идея

Основаная идея *so_alt_channel* (выраженная в интерфейсе *cyclic_connector_controller_t*) заключается в том, что вместо стандартного connector_controller-а транспортному агенту подсовывается *cyclic_connector_controller*. Когда транспортный агент обращается к методу *connect()* этого connector_controller-а, *cyclic_connector_controller* пытается установить соединение сначала по самому приоритетному адресу, затем по менее приоритетному, затем по еще менее приоритетному и т.д. Если соединение по какому-то подключению установлено, то транспортный агент

будет использовать именно его до тех пор, пока соединение по каким-то причинам не будет закрыто. При закрытии соединения транспортный агент вновь обращается к методу *connect()* и *cyclic_connector_controller* повторят эту процедуру заново.

Для того, чтобы не выполнять циклический перебор слишком часто *cyclic_connector_controller* запоминает время установления последнего соединения. Если при очередном вызове *connect()* оказывается, что с момента установления соединения прошло слишком мало времени, то *cyclic_connector_controller* считает, что соединение было неудачным, и продолжает циклический перебор не с самого начала, а со следущего за этим проблемным подключением адреса.

12.3 Способ использования

Для использования *cyclic_connector_controller* программисту необходимо:

1. Создать экземпляр *cyclic_connector_controller*.
2. Для каждого альтернативного адреса создать необходимый конкретный connector_controller (например, для TCP/IP соединений это выполняется с помощью функций *so_4::transport_layer::socket::create_connector_controller()*.
3. Зарегистрировать в *cyclic_connector_controller* все конкретные connector_controller-ы в порядке убывания их приоритетов (т.е. connector_controller для основного подключения должен идти первым).
4. Передать подготовленный таким образом экземпляр *cyclic_connector_controller* транспортному агенту.

12.4 Пример использования

В качестве примера использования *so_alt_channel* используется код фабрики кооперациий для исходящих каналов из *so_sysconf* (6.2 на стр. 41):

```
1  /*
2    SoSysconf 2
3  */
4
5  /*!
6    \since v.2.3.0
7    \file
8    \brief Фабрика кооперациий ochannel-ов.
9  */
10
11 #include <algorithm>
12 #include <functional>
13
14 #include <cpp_util_2/h/lexcast.hpp>
15
16 #include <so_4/rt/h/rt.hpp>
17 #include <so_4/rt/comm/h/a_sop_outgoing_channel.hpp>
18
19 #include <so_4/transport_layer/socket/h/pub.hpp>
20
```

```
21 #include <so_4/sop/h/filter.hpp>
22
23 #include <so_alt_channel_2/h/pub.hpp>
24
25 #include <so_sysconf_2/h/coop_factory.hpp>
26 #include <so_sysconf_2/h/app_paths.hpp>
27 #include <so_sysconf_2/h/a_trouble.hpp>
28
29 #include <so_sysconf_2_ochannel/h/cfg.hpp>
30
31 namespace so_sysconf_2
32 {
33
34 namespace ochannel
35 {
36
37 //
38 // coop_factory_t
39 //
40
41 /*!
42   \since v.2.3.0
43   \brief Фабрика кооперации ochannel-ов.
44
45   \par Псевдоним DLL
46   so_sysconf_2::ochannel
47
48   \par Имя фабрики
49   so_sysconf_2::ochannel::factory
50 */
51 class coop_factory_t
52   : public so_sysconf_2::coop_factory_t
53   {
54     //! Псевдоним для базового типа.
55     typedef so_sysconf_2::coop_factory_t  base_type_t;
56   public :
57     coop_factory_t()
58       : base_type_t(
59           "so_sysconf_2::ochannel",
60           "so_sysconf_2::ochannel::factory" )
61       {}
62
63     //! Регистрация новой кооперации.
64     /*!
65       Невозможность регистрации кооперации является фатальной ошибкой.
66     */
67     virtual bool
68     reg(
69       const std::string & coop_name,
70       const std::string & cfg_file,
71       std::string & error_msg );
72   };
73
74 //! Предикат для std::for_each.
```

```
75 /*!
76   Добавляет в SOP-фильтр имя очередного агента.
77 */
78 class filter_filler_t
79   : public std::unary_function< const std::string &, void >
80   {
81     so_4::sop::std_filter_t & m_filter;
82   public :
83     filter_filler_t(
84       so_4::sop::std_filter_t & filter )
85       : m_filter( filter )
86       {}
87
88     result_type
89     operator()( argument_type a )
90       {
91         m_filter.insert( a );
92       }
93   };
94
95 //! Предикат для std::for_each.
96 /*!
97  * Создает connector_controller для очередного IP-узла и добавляет
98  * его к указанному cyclic_connector_controller-у.
99  */
100 class connector_controller_appender_t
101   : public std::unary_function< const std::string &, void >
102   {
103     so_alt_channel_2::cyclic_connector_controller_t & m_owner;
104     const so_4::transport_layer::channel_params_t & m_channel_params;
105   public :
106     connector_controller_appender_t(
107       so_alt_channel_2::cyclic_connector_controller_t & owner,
108       const so_4::transport_layer::channel_params_t & channel_params )
109       : m_owner( owner )
110       , m_channel_params( channel_params )
111       {}
112
113     result_type
114     operator()( argument_type a )
115       {
116         using namespace so_4::transport_layer;
117         using namespace so_4::transport_layer::socket;
118
119         m_owner.add_connector_controller(
120             create_connector_controller(
121                 connector_params( a ),
122                 m_channel_params ) );
123       }
124   };
125
126
127 //! Создать кооперацию агентов.
128 std::auto_ptr< so_4::rt::dyn_agent_coop_t >
```

```
129 make_coop(
130   //! Имя, которое должна иметь кооперация.
131   const std::string & coop_name,
132   //! Конфигурация кооперации.
133   const cfg_t & cfg )
134   {
135     std::string channel_agent_name =
136         ( cfg.channel_agent_name().empty() ?
137             // Т.к. имя в конфиге не задано, то создаем его сами.
138             coop_name + "::a_channel" :
139             cfg.channel_agent_name() );
140
141     // Создаем фабрики для работы по альтернативным каналам.
142     so_alt_channel_2::cyclic_connector_controller_auto_ptr_t connector(
143         so_alt_channel_2::create_cyclic_connector_controller(
144             cfg.minimal_connection_time() ) );
145     std::for_each(
146         cfg.addresses().begin(),
147         cfg.addresses().end(),
148         connector_controller_appender_t(
149             *connector,
150             cfg.channel_params() ) );
151
152     // Для канала должен быть подготовлен специальный фильтр.
153     std::auto_ptr< so_4::sop::std_filter_t > filter(
154         so_4::sop::create_std_filter() );
155     std::for_each(
156         cfg.filter_agent_list().begin(),
157         cfg.filter_agent_list().end(),
158         filter_filler_t( *filter ) );
159
160     std::auto_ptr< so_4::rt::comm::a_sop_outgoing_channel_t > a_channel(
161         new so_4::rt::comm::a_sop_outgoing_channel_t(
162             channel_agent_name,
163             so_4::transport_layer::connector_controller_auto_ptr_t(
164                 connector ),
165             so_4::sop::filter_auto_ptr_t( filter ),
166             so_4::rt::comm::create_def_disconnect_handler(
167                 cfg.reconnect_timeout() * 1000,
168                 cfg.restore_timeout() * 1000 ) ) );
169     a_channel->set_handshaking_params( cfg.handshaking_params() );
170
171     so_4::rt::agent_t * coop_agents[] =
172         {
173           a_channel.release()
174         };
175
176     std::auto_ptr< so_4::rt::dyn_agent_coop_t > coop(
177         new so_4::rt::dyn_agent_coop_t(
178             coop_name,
179             coop_agents,
180             sizeof( coop_agents ) / sizeof( coop_agents[ 0 ] ) ) );
181
182     return coop;
```

```
183   }
184
185 bool
186 coop_factory_t::reg(
187   const std::string & coop_name,
188   const std::string & cfg_file,
189   std::string & error_msg )
190   {
191     cfg_t cfg;
192     if( load_cfg_file(
193         so_sysconf_2::app_paths_t::instance()->etc_file_name( cfg_file ),
194         cfg, error_msg ) )
195       {
196         so_4::rt::dyn_agent_coop_helper_t helper(
197             make_coop( coop_name, cfg ).release() );
198         if( !helper.result() )
199           // Кооперация успешно создана.
200           return true;
201         else
202           error_msg = "unable to register coop" +
203               cpp_util_2::slexcast( helper.result() );
204       }
205
206     so_sysconf_2::a_trouble_t::send_msg_fatal_error(
207       "so_sysconf_2_ochannel::coop_factory_t::reg()",
208       "unable_to_register_coop",
209       error_msg );
210
211     return false;
212   }
213
214 //! Этот объект и будет фабрикой.
215 coop_factory_t  g_factory;
216
217 } /* namespace ochannel */
218
219 } /* namespace so_sysconf_2 */
```

Созданием кооперации, в которую входит транспортный и управляющий агенты, занимается функция *make_coop* (строки 128–183). В строках 142–144 создается *cyclic_connector_controller*, который в строках 145-150 заполняется конкретными connector_controller-ами. Сами же connector_controller-ы для TCP/IP соединений создаются в функторе *connector_controller_appender_t* (строки 100–124).

В строках 153–158 формируется фильтр коммуникационного канала, в котором перечисляются имена всех агентов, указанные в конфигурационном файле. При этом используется вспомогательный класс-предикат *filter_finder_t* определенный в строках 78–93.

В строке 160 начинается создание транспортного агента, которому в качестве connector_controller-а передается экземпляр *cyclic_connector_controller*. А так же назначается disconnect_handler, который будет отвечать за восстановление соединения в случае разрывов связи, — строки 166–168.

Часть VI

Message Box API

Приложение А

Принципы работы коммуникационных каналов SObjectizer

Коммуникационные каналы SObjectizer используют описанную ниже схему работы.

Для обнаружения входящих данных в канале SObjectizer средствами ОС прослушивает канал. Когда SObjectizer обнаруживает, что в канале есть данные, он вычитывает из канала пакет. Максимальный размер этого пакета задается значением *input_portion_size* и по умолчанию равен 32K. На практике, из канала может быть прочитано и меньше данных — это определяется уже транспортным уровнем ОС. Для входящих данных есть т.н. порог входящих данных *input_threshold*. По умолчанию порог равен 1000 пакетов (*package_count*) или 100Kb трафика (*traffic_bulk*) – в зависимости от того, что раньше произойдет. При каждом успешном чтении данных из канала SObjectizer проверяет, не превышен ли порог входящих данных. Если превышен, то канал объявляется блокированными и чтение из него прекращается до тех пор, пока канал не разблокируется. Разблокировка выполняется SObjectizer-ом автоматически по мере разбора прочитанных из канала данных. А уже это определяется общей загрукой приложения. Время, которое канал может провести в заблокированном состоянии ограничено (праметр *max_input_block_timeout*, по умолчанию 30 секунд). Если канал находится в заблокированном состоянии слишком долго, то он принудительно закрывается.

Для эффективной отсылки исходящих данных для каждого канала SObjectizer поддерживает два буфера — *awaiting_buffer* и *outgoing_buffer*. Первый используется для накопления данных, второй — для записи данных в канал. При появлении очередного исходящего сообщения его двоичный образ помещается сначала в *awaiting_buffer*, затем, непосредственно перед отправкой — в *outgoing_buffer*. Запись в канал ведется блоками. Максимальный размер блока определяется параметром *output_portion_size*. По умолчанию он равен 32K, но может быть и меньше, если в *output_buffer/awaiting_buffer* готовых к отправке данных меньше.

Изначально размеры *awaiting_buffer* и *output_buffer* равны *output_portion_size*, но могут увеличиваться до величины, заданной параметром *max_awaiting_buffer_size/max_output_buffer_size* (по умолчанию — 512Kb). Если исходящие данные формируются в SObjectizer быстрее, чем успевают отсылаться в канал, то один их этих буферов переполняется и канал принудительно закрывается.

SObjectizer выполняет попытки записи в двух случаях:

1. При появлении исходящего сообщения, когда *output_buffer* пуст. Т.е. SObjectizer ждал

исходящих данных, они появились и SObjectizer сразу же записал их в канал;

2. Когда ОС говорит, что канал готов к записи.

Во втором случае ситуация следующая: появились исходящие данные, SObjectizer поместил их в *outgoing_buffer*, записал порцию в канал. И здесь может оказаться, что либо в *outgoing_buffer* было больше данных, чем *output_portion_size*, либо же в канал записалось меньше данных (что определяется транспортным уровнем ОС). Т.е. после записи в канал в *output_buffer* остались еще исходящие данные. SO-4.4.0-b5 не пытается их сразу же дописывать, т.к. это может привести к перегрузке транспортного канала и, возможно, блокировки на некоторое время SObjectizer. Вместо этого SObjectizer средствами ОС начинает контролировать готовность канала к операции записи. Если канал готов — SObjectizer пишет очередную порцию данных. Если не готов (а это может произойти, если удаленная сторона не успевает вычитывать данные из канала), то SObjectizer ждет готовности.

Максимальное время ожидания задается параметром *max_output_block_timeout* (по умолчанию 30 секунд). Если канал за это время в состояние готовности не перешел, то он принудительно закрывается.

SObjectizer проверяет время блокированности чтения из канала и время не готовности канала к записи периодически. Период проверки задается параметром *time_checking_period* (по умолчанию 1 секунда).

Для исходящих данных SObjectizer использует два буфера потому, что SOP-протокол поддерживает возможность трансформации трафика, например, архивирование трафика с помощью zip-сжатия (т.н. zip-ование). Поэтому в *outgoing_buffer* сохраняются полностью готовые к отправке данные (т.е. подвершиеся трансформации). В *awaiting_buffer* — еще не готовые (т.е. им еще предстоит пройти через трансформацию).

Если zip-ование трафика не выполняется, то исходящие данные сразу формируются в *outgoing_buffer*. Но, если используется zip-ование, то сначала данные накапливаются в *awaiting_buffer* и только затем, по мере освобождения *outgoing_buffer*, архивируются из *awaiting_buffer* в *outgoing_buffer*. Такая схема позволяет при большом количестве исходящих сообщений сначала скопить их в *awaiting_buffer*, затем заархивировать в *outgoing_buffer* и отослать одним небольшим пакетом.

Все вышеописанные параметры называются параметрами канала (*channel_params*).

Когда SOP-клиент подключается к SOP-серверу, он выполняет специальную процедуру *handshake*, во время которой стороны определяют параметры передачи данных (данные параметры называются *handshaking_params*). В настоящее время таким параметром является только компрессия (т.е. zip-ование трафика). Если и клиенту, и серверу разрешено использовать компрессию, то они начинаю zip-овать трафик. Если же хотя бы одному из них компрессия запрещена, то zip-ование не выполняется.

По умолчанию в SO-4.4.0-b4 и SO-4.4.0-b5 компрессия запрещена (более того, в SO-4.4.0-b4 она вообще не поддерживается). В SO-4.4.0-b5 компрессию можно включить управляя параметрами процедуры handshake.

Литература

[SO4] http://sobjectizer.sourceforge.net

[CPPD] Страуструп Б. Дизайн и эволюция C++: Пер.с англ. – М.:ДМК Пресс; Спб.:Питер, 2006.

[ATUP] Реймонд, Эрик С. Искусство программирования для Unix: Пер.с англ. – М.:Издательский дом "Вильямс 2005.

[BDB] http://www.sleepycat.com/products/bdb.html.

[SQLITE] http://www.sqlite.org/.

[L4J] http://logging.apache.org/log4j/docs/.

[SOBOOK] Е. Охотников. SObjectizer-4 Book, 2004. http://eao197.narod.ru/desc/so_4_book.pdf

[OESS] http://eao197.narod.ru/objessty

[ACE] http://www.cs.wustl.edu/~schmidt/ACE.html

[WRAPM] B. Stroustrup. Wrapping C++ Member Function Calls. The C++ Report. June 2000, Vol 12/No 6. http://www.research.att.com/~bs/wrapper.pdf