Построенные над SObjectizer библиотеки

Евгений Охотников
Intervale

4 августа 2006 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

Библиотека *so_alt_channel* содержит специальную реализацию обработчика разрывов TCP/IP соединений. Штатный обработчик, создаваемый методом `create_def_disconnect_handler()` класса `a_cln_channel_base_t`, реализует тривиальную логику: периодические переподключения по тому же адресу после тайм-аута. В некоторых случаях при разрыве основного TCP/IP подключения требуется попробовать подключиться по одному из резервных адресов. И, если это удалось, работать по резервному каналу. Но повторяя при этом попытки восстановить подключение по основному каналу. Как только подключение по основному каналу восстановлено, резервное соединение должно быть закрыто, а дальнейшая работа должна вестись по основному соединению.

В библиотеке *so_alt_channel* находится реализация подобной логики поведения и несколько вспомогательных агентов, которые упрощают ее внедрение в SObjectizer-приложение.

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. 34).

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 кооперацией. Он предназначен для случаев, когда заранее не известно, сколько кооперацией требуется создать в приложении и какие имена будут иметь эти кооперации.

Как и *coop_handler*, *coop_factory* выполнение двух операций: регистрацию и дерегистрацию кооперации. Но, поскольку имена кооперацией 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 коопераций доступны в данной DLL (при этом используется псевдоним DLL). Таким образом *so_sysconf* узнает о существовании коопераций и фабрик коопераций. Соответственно, при выгрузке DLL (после получения сообщения `msg_unload_dll`) деструкторы *coop_handler* и *coop_factory* вычеркивают имена коопераций 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. 34) 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(
```

```
5     // Имя файла DLL. Должно задаваться конкретное имя с учетом
6     // особенностей конкретной ОС.
7     "./sample.dll",
8     // Уникальный псевдоним для DLL.
9     "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 кооперaций.
    "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 достаточно сложной и не гибкой. 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:

```
1 ||
2 || Скрипт начальной инициализации AAG 3
3 ||
4 {sysconf-script
5
6   |#
7     Загрузка и инициализация средств обработки прерывания приложения.
8     Содержит кооперации:
9       so_sysconf_2::breakflag_handler::user_break_handler
10      so_sysconf_2::breakflag_handler::system_break_handler
11  #|
```

```
12  {load-dll "so_sysconf.breakflag_handler"
13    {alias  "so_sysconf_2::breakflag_handler"}
14    {os-name-convert  "simple" }
15  }
16  || Обработчик прерывания приложения операционной системой.
17  {reg-coop "so_sysconf_2::breakflag_handler::system_break_handler"
18  }
19  || Обработчик прерывания приложения пользователем.
20  {reg-coop "so_sysconf_2::breakflag_handler::user_break_handler"
21  }
22
23  |#
24    Подсистема so_log_1.
25  #|
26  {load-dll "so_sysconf_log.sysconf"
27    {alias  "so_sysconf_log_1::sysconf" }
28    {os-name-convert  "simple" }
29  }
30  {reg-coop "so_sysconf_log_1::sysconf::log" }
31
32  |#
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 }
```

Не трудно заметить, 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. 20).

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 кооперций через конфигурационный файл приведен в 4.1.2 на стр. 17.

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:

```
1 bool
2 coop_factory_t::reg(
3   const std::string & coop_name,
4   const std::string & cfg_file,
5   std::string & error_msg )
6 {
7   cfg_t cfg;
8   if( load_cfg_file(
9       // Вот использование app_paths для получения полного
10      // имени конфигурационного файла.
11      so_sysconf_2::app_paths_t::instance()->etc_file_name( cfg_file ),
12      cfg,
13      error_msg ) )
14  {
15    std::auto_ptr< so_4::rt::agent_t > a_bufferizator(
16        new a_bufferizator_t( coop_name + "::a_bufferizator", cfg ) );
17
```

```
18      so_4::rt::agent_t * main_coop_agents[] =
19      {
20        a_bufferizator.release()
21      };
22
23      // Регистрируем кооперацию.
24      so_4::rt::dyn_agent_coop_helper_t coop_helper(
25        new so_4::rt::dyn_agent_coop_t(
26          coop_name, main_coop_agents, 1 ) );
27      if( !coop_helper.result() )
28      {
29        // Кооперация успешно зарегистрирована.
30        so_log_1::logic[ so_log_1::normal ]
31            [ so_log_1::a() << query_factory_name() ]
32            [ so_log_1::n() << "factory::created" ]
33            [ so_log_1::d() << "coop_name: " << coop_name
34                << ", cfg_file: " << cfg_file ]();
35        return true;
36      }
37      else
38      {
39        error_msg = std::string( "unable to register coop: " ) +
40          coop_helper.result().m_desc;
41      }
42    }
43
44    // 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. 42).

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 кооперaций.
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_srv_channel.hpp>
22 
23 #include <so_4/socket/channels/h/channels.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::a_srv_channel_base_t::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-a
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::a_srv_channel_base_t::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( "evt_comm_socket_creation_fail",
115       m_a_socksrv_name, "msg_fail" );
116 }
117
118 void
119 a_failure_handler_t::evt_comm_socket_creation_fail(
120   const so_4::rt::comm::a_srv_channel_base_t::msg_fail & cmd )
121 {
122   // Это фатальная ошибка!
123   so_sysconf_2::a_trouble_t::send_msg_fatal_error(
124     so_query_name(),
125     "unable_to_create_server_socket",
126     cpp_util_2::slexcast( cmd.m_ret_code ) );
127 }
128
129 //
130 // coop_factory_t
131 //
132
133 std::string
134 coop_factory_t::m_dll_alias( "so_sysconf_2::ichannel" );
135
136 std::string
137 coop_factory_t::m_factory_name( "so_sysconf_2::ichannel::factory" );
138
139 coop_factory_t::coop_factory_t()
140   :
141     base_type_t( m_dll_alias.c_str(), m_factory_name.c_str() )
142 {
143 }
```

```
144
145 coop_factory_t::~coop_factory_t()
146 {
147 }
148
149 bool
150 coop_factory_t::reg(
151   const std::string & coop_name,
152   const std::string & cfg_file,
153   std::string & error_msg )
154 {
155   cfg_t cfg;
156   if( load_cfg_file(
157       app_paths_t::instance()->etc_file_name( cfg_file ),
158       cfg, error_msg ) )
159   {
160     // Имя агента-сокета в кооперации.
161     std::string a_socksrv_name( cfg.m_channel_agent_name );
162     if( a_socksrv_name.empty() )
163       a_socksrv_name = coop_name + "::a_ichannel";
164
165     // Коммуникационный сокет для входа в приложение.
166     std::auto_ptr< so_4::rt::agent_t > a_socksrv(
167         new so_4::rt::comm::a_srv_channel_t( a_socksrv_name,
168             so_4::socket::channels::create_server_channel(
169                 cfg.m_ip ) ) );
170     // Разберемся, на какой нити будет работать транспортный агент.
171     if( cfg.m_is_active_obj )
172       so_4::disp::active_obj::make_active( *a_socksrv );
173     else if( !cfg.m_active_group_name.empty() )
174       so_4::disp::active_group::make_member( *a_socksrv,
175           cfg.m_active_group_name );
176
177     // Контроллер невозможности создания серверного сокета.
178     std::auto_ptr< so_4::rt::agent_t > a_failure_handler(
179         new a_failure_handler_t(
180             coop_name + "::a_failure_handler",
181             a_socksrv_name ) );
182
183     so_4::rt::agent_t * main_coop_agents[] =
184     {
185       a_socksrv.release(), a_failure_handler.release()
186     };
187
188     // Регистрируем кооперацию.
189     so_4::rt::dyn_agent_coop_helper_t coop_helper(
190       new so_4::rt::dyn_agent_coop_t(
191         coop_name.c_str(), main_coop_agents, 2 ) );
192     if( !coop_helper.result() )
193       return true;
194
195     error_msg = std::string( "unable to register coop: " ) +
196       coop_helper.result().m_desc;
197   }
```

```
198
199   // Это фатальная ошибка!
200   so_sysconf_2::a_trouble_t::send_msg_fatal_error(
201     "so_sysconf_2_ichannel::coop_factory_t::reg()",
202     "unable_to_register_coop",
203     error_msg );
204
205   return false;
206 }
207
208 //! Этот объект отвечает за фабрику.
209 coop_factory_t g_coop_factory;
210
211 } /* namespace ichannel */
212
213 } /* namespace so_sysconf_2 */
```

Агент `a_failure_handler` необходим для того, чтобы получить результат создания серверного сокета агентом типа *so_4::rt::comm::a_srv_channel_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. 23). 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. 34).

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`.

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_srv_channel_t*), которые отвечают за обслуживание серверных TCP/IP сокетов, и клиентские (класс *so_4::rt::comm::a_srv_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_srv_channel_t*;
- `so_sysconf.ochannel.sysconf` (псевдоним `so_sysconf_2::ochannel`), в которой находится фабрика `so_sysconf_2::ichannel::factory` для создания агентов типа *so_4::rt::comm::a_srv_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 находящаяся 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. 23). 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>}]
}
```

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.

Ошибка в конфигурационном файле или невозможность создания серверного сокета с указанным IP-адресом приводит к порождению фатальной ошибки (см. 4.3 на стр. 21).

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. 23). 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> ...] }
}
```

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. 50). 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 периодически транспортный агент пытается установить соединение по основному адресу (либо по одному из более приоритетных резервных адресов). Если это удается, то текущее соединение закрывается, а транспортный агент переходит на работу с только что установленным более приоритетным соединением.

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.

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. 21).

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.
  }
```

```
6
7   {reconnect_timeout 20 }
8   {restore_timeout 0 }
9
10  {filter_agent_list "mbapi_3::a_mbapi" }
11 }
```

Часть III

Generic Monitoring Tool

Часть IV

SObjectizer Logger

Часть V

SObjectizer Alternative Channel

Глава 7

Введение

7.1 Описание проблемы

Идея каналов ввода-вывода в SObjectizer базируется на том, что каждый канал предназначен для соединения с одним конкретным узлом. Поскольку клиентские транспортные агенты в SObjectizer (*so_4::rt::comm::a_cln_channel_t*, *so_4::rt::comm::a_raw_cln_channel_t*) работают с одним каналом, то и они предназначены для работы с одним узлом. Т.е. если соединение с удаленной стороной есть, то все хорошо. 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 это сложно. Ведь мало того, что нужно отслеживать сообщения `msg_client_connected`, `msg_client_disconnected` от нескольких агентов, так еще нужно контролировать, какой агент потерял соединение, и какого агента нужно заставить установить новое соединение.

Более удобен вариант создания специального класса, реализующего интерфейс *so_4::rt::comm::client_factory_t*. Такой класс мог бы получать в конструкторе список реальных фабрик, по одной на каждый канал. В методе *connect()* этот класс последовательно перебирал бы все фабрики, пока не установил бы соединение.

Данный вариант хорош, если устраивает вариант последовательного перебора подключений. Т.е. если теряется связь на основном подключении, то происходит переход на резервное подключение и работа ведется на резервном подключении до тех пор, пока не оно не будет разорвано. После разрыва резервного подключения происходит возврат на основное и т.д.

Но что делать, если требуется при работе по резервному подключению постоянно контролировать жизнеспособность основного подключения? И как только основное подключение приходит в норму, то нужно закрыть резервное подключение? Сложностей здесь две:

- подобная фабрика является пассивным объектом, не агентом. Она не может выделять время сама себе для периодических попыток

восстановления основного канала;

- в случае SOP-соединений резервное подключение нужно закрыть как только будет обнаружено восстановление физического канала на основном подключении. Т.е. нельзя дать SObjectizer осуществить установку логического SOP-соединения на основном канале, пока не будет закрыт резервный канал.

Проект *so_alt_channel* как раз предоставляет описанную реализацию интерфейса *so_4::rt::comm::client_factory_t* и набор средств для преодоления двух вышеуказанных проблем.

7.2 Основная идея

Идея состоит в том, что для работы с альтернативными каналами требуются:

- реализация интерфейса *so_4::rt::comm::client_factory_t*, которая позволяет последовательно перебирать фабрики при установке соединения;
- агент, который бы периодически требовал у фабрики попытаться восстановить основное соединение.

В качестве реализации интерфейса *so_4::rt::comm::client_factory_t* *so_alt_channel* предлагает класс *so_alt_channel_1::client_factory_t* с простым принципом работы. В конструктор передается список реальных фабрик в виде объекта *so_alt_channel_1::factories_t*. Все фабрики упорядочены в виде списка. Первый элемент списка с индексом 0 является «основным подключением». Все остальные элементы списка являются «резервными подключениями». Причем приоритет подключения уменьшается с увеличением индекса. Задачей *client_factory_t* является создание канала на подключении с минимальным индексом.

Для этого в методе *so_alt_channel_1::client_factory_t::connect()* осуществляется последовательный проход по списку фабрик от первого элемента к последнему. Возвращается первый успешно созданный канал. При этом в объекте *client_factory_t* сохраняется индекс подключения. Если затем обратиться к методу *so_alt_channel_1::client_factory_t::try_switch()*, то будет осуществлен проход по списку фабрик до этого индекса. Если с помощью какой-то фабрики будет успешно создан канал, значит удалось установить соединение по более приоритетному подключению. Созданный объект *so_4::rt::comm::io_channel_t* сохраняется внутри *client_factory_t* и возвращается при следующем вызове метода *connect()*.

Объекту *so_alt_channel_1::client_factory_t* в конструкторе должно быть указано имя агента, владеющего сообщениями `msg_channel_switched` (реализуется типом *so_alt_channel_1::msg_channel_switched*) и `msg_switch_required` (реализуется типов *so_alt_channel_1::msg_switch_required*). Когда в методе *client_factory_t::connect()* успешно создается канал, то отсылается сообщение `msg_channel_switched` этого агента. В отосланном

сообщении содержится индекс подключения, на котором создан канал. Когда канал создается в методе *client_factory_t::try_switch()*, то отсылается сообщение `msg_switch_required`, в котором так же содержится индекс успешного подключения.

Сообщениями `msg_channel_switched` и `msg_switch_required` может владеть любой агент. Более того, имена этих сообщений могут быть произвольными (они задаются в конструкторе *client_factory_t*). Но для удобства, *so_alt_channel* предоставляет уже готового агента `so_alt_channel_1::a_helper_t`, который не только является владельцем указанных сообщений, но еще периодически вызывает метод *client_factory_t::try_switch()*.

Глава 8

Использование

8.1 Основной принцип использования

Для использования *so_alt_channel* необходимо иметь транспортного агента, которому в качестве фабрики назначается объект *so_alt_channel_1::client_factory_t*, обработчик разрывов связи и управляющего агента.

Задачей управляющего агента является отслеживание сообщений `msg_client_connected` от транспортного агента. Получив сообщение `msg_client_connected` управляющей агент должен сохранить у себя идентификатор канала, т.к. он будет нужен в дальнейшем.

Управляющий агент должен обрабатывать сообщения `msg_switch_required`, которые отсылает *client_factory_t*. Получив это сообщение управляющий агент должен инициировать закрытие текущего соединения отсылкой сообщения `msg_close_channel` с ранее сохраненным идентификатором канала. Когда транспортный агент по сообщению `msg_close_channel` закроет текущее соединение сработает обработчик разрывов связи и заставит транспортного агента попытаться восстановить соединение. Для чего транспортный агент обратиться к *client_factory_t::connect()* и получит канал, созданный при обращении к *client_factory_t::try_switch()*.

Какой-то агент, может быть, управляющий, должен периодически вызывать метод *client_factory_t::try_switch()*. Этот метод достаточно интеллектуален и его можно вызывать независимо о того, если ли сейчас соединение или нет. Он будет работать только, если соединение есть, и установлено оно не по основному подключению.

В качестве агента, который будет постоянно вызывать метод *client_factory_t::try_switch()*, удобно использовать агента `so_alt_channel_1::a_helper_t`. Агент `so_alt_channel_1::a_switch_enforcer_t` кроме этого способен самостоятельно инициировать смену текущего канала, т.е. способен быть полноценным управляющим агентом.

8.2 Вспомогательные агенты

8.2.1 Агент a_helper_t

Агент `so_alt_channel_1::a_helper_t` — это вспомогательный агент для владения *so_alt_channel* сообщениями и периодическими попытками переключения каналов.

Владеет сообщениями:

msg_channel_switched (тип *so_alt_channel_1::msg_channel_switched*);

msg_switch_required (тип *so_alt_channel_1::msg_switch_required*);

msg_try_switch (тип *so_alt_channel_1::a_helper_t::msg_try_switch*).

Сообщение `msg_try_switch` является периодическим, его темп задается в конструкторе. Обработчик `msg_try_switch` просто обращается к *client_factory_t::try_switch()*.

Т.к. данный агент будет выполнять операции ввода-вывода, которые могут оказаться достаточно длительными, то желательно, чтобы данный агент имел собственную рабочую нить. Например, был активным агентом или входил в активную группу.

Примечание. Данный агент только помогает определить, что возможно переключение на более приоритетный коммуникационный канал, но он не инициирует это переключение. Предполагается, что `a_helper_t` будет использоваться в связке с прикладным агентом. И уже прикладной агент будет обрабатывать сообщение `msg_switch_required` и именно прикладной агент будет отвечать за закрытие текущего коммуникационного канала и переподключение на более приоритетный канал.

8.2.2 Агент a_switch_enforcer_t

Этот агент является производным от агента `a_helper_t`. Главное отличие в том, что `a_switch_enforcer_t` предназначен для контроля за состоянием транспортного агента и за инициирование переподключения транспортного агента после того, как будет восстановлено соединение по более приоритетному каналу.

Для выполнения своей задачи агент `a_switch_enforcer` подписывается на сообщение `msg_client_connected` транспортного агента и в обработчике сообщения сохраняет идентификатор канала.

Так же агент `a_switch_enforcer_t` подписывается на собственное сообщение `msg_switch_required` и в его обработчике отсылает транспортному агенту сообщение `msg_close_channel`.

В отличии от `a_helper_t` агент `a_switch_enforcer_t` может играть роль самостоятельного управляющего агента. Т.е., например, можно создать кооперацию, в которую будут входить транспортный агент, прикладной агент и агент `a_switch_enforcer_t`. При это прикладному агенту не нужно будет заботится о контроле транспортного агента — за все детали будет отвечать `a_switch_enforcer_t`.

8.3 Особенности

8.3.1 Разрушение фабрики

Транспортные агенты получают в конструкторе указатель на динамически созданный объект *so_4::rt::comm::client_factory_t* и уничтожают его в своем деструкторе. В случае с использованием *so_alt_channel_1::client_factory_t* это является источником опасности.

Передавать указатель на объект *so_alt_channel_1::client_factory_t* в конструктор транспортных агентов можно только, если никто не использует больше этот объект. Но, кто-то же должен вызывать у client_factory метод *try_switch()*! Как правило, это еще один агент, управляющий или вспомогательный. Т.е. получается, что указатель на client_factory будет использоваться несколькими агентами одновременно. Поэтому не желательно, чтобы транспортный агент уничтожил client_factory в своем деструкторе.

Если so_alt_channel_1::client_factory_t используется несколькими агентами, то нужно придерживаться следующих правил:

- транспортному агенту передается указатель на объект *so_alt_channel_1::proxy_t*, который не уничтожает client_factory в своем деструкторе;
- объект *so_alt_channel_1::client_factory_t* лучше всего сохранять как объект, контролируемый кооперацией.

Например:

```
1 // Создаем хитрую фабрику.
2 // Остальные фабрики уже созданы и помещены в хранилище factories.
3 so_alt_channel_1::client_factory_t * f =
4   new so_alt_channel_1::client_factory_t(
5     factories,
6     // Псевдоним для канала. Может помочь управляющему
7     // агенту, если он работает сразу с несколькими такими каналами.
8     "my_channel",
9     // Агент, который владеет сообщениями msg_channel_switched,
10    // msg_switch_required.
11    "a_helper" );
12
13 // Создаем агентов, которые будут входить в нашу кооперацию.
14
15 // Прикладной агент, который отвечает за реальную работу.
16 a_handler_t * a_handler = new a_handler_t(
17   "a_handler", "a_channel", "a_helper" );
18
19 // Транспортный агент.
20 so_4::rt::comm::a_cln_channel_t * a_channel =
21   new so_4::rt::comm::a_cln_channel_t(
22     "a_channel",
23     // Назначаем прокси вместо реальной фабрики.
24     so_alt_channel_1::proxy_t::create( *f ),
25     so_4::sop::create_all_enable_filter(),
```

```
26    // Обработчик разрывов связи.
27    so_4::rt::comm::a_cln_channel_base_t::
28      create_def_disconnect_handler( 5000, 1000 ) );
29
30 // Вспомогательный агент, который владеет сообщениями
31 // msg_channel_switched, msg_switch_required и периодически
32 // вызывает метод try_switch у фабрики.
33 so_alt_channel_1::a_helper_t *  a_helper =
34   new so_alt_channel_1::a_helper_t(
35     "a_helper", *f,
36     // Повторяем попытки каждые три секунды.
37     3000 );
38
39 // Создаем кооперацию.
40 so_4::rt::agent_t * agents[] =
41   {
42     &a_handler
43   , &a_channel
44   , &a_helper
45   };
46 so_4::rt::dyn_agent_coop_t * coop =
47   new so_4::rt::dyn_agent_coop_t(
48       "sample_sop",
49       agents,
50       sizeof( agents ) / sizeof( agents[ 0 ] ) );
51 so_4::rt::dyn_coop_controlled( *coop, f );
```

8.3.2 Приоритеты обработчиков событий

Управляющий агент должен обрабатывать сообщения `msg_client_connected` и `msg_switch_required` с одинаковым приоритетом. Обработчики сообщений `msg_client_connected`, `msg_switch_required` должны быть либо normal-, либо insend-событиями одновременно. Нужно это для того, чтобы обеспечить обработку этих сообщений в строгой хронологической последовательности. В противном случае может произойти следующая ситуация:

1. *`client_factory_t`* пытается восстановить основное подключение и это удается. Созданный io_channel сохраняется в client_factory до последующего обращения к *`connect()`*. client_factory отсылает сообщение `msg_switch_required`.

2. В этот момент диагностируется разрыв связи на текущем подключении. Обработчик разрывов связи принуждает транспортного агента вызвать метод *`client_factory_t::connect()`* и метод *`connect()`* возвращает уже созданный объект io_channel. Транспортный агент отсылает сообщение `msg_client_connected`.

3. Если управляющий агент обрабатывает `msg_client_connected` с большим приоритетом, чем `msg_switch_required` (либо обрабатывает `msg_client_connected` как insend-событие), то он получит `msg_client_connected` до `msg_switch_required`. И когда поступит

`msg_switch_required` управляющий агент прикажет транспортному агенту закрыть только что установленное соединение!

8.3.3 Использование обработчика разрывов связи

Использование обработчиков разрывов связи является чрезвычайно важным. Весь механизм основан на том, что когда управляющий агент отсылает транспортному агенту сообщение `msg_close_channel`, транспортный агент автоматически попытается восстановить соединение. Но транспортный агент просто так этого не делает. Его заставляет сделать это обработчик разрывов связи. Поэтому транспортному агенту, который работает с *so_alt_channel_1::proxy_t* (*so_alt_channel_1::client_factory_t*) очень желательно назначать обработчик разрывов связи.

Если этого не сделать, то к задачам управляющего агента добавляется так же отсылка сообщения `msg_connect` транспортному агенту после того, как будет отослано сообщение `msg_close_channel`.

8.3.4 Использование логгера

Иногда бывает необходимо, чтобы управляющий агент мог залогировать результаты попыток установления/восстановления соединения, которые предпринимаются в методах *so_alt_channel_1::client_factory_t::connect()*, *so_alt_channel_1::client_factory_t::try_switch()*. Для этого может использоваться интерфейс *so_alt_channel_1::logger_t*: нужно создать объект, реализующий данный интерфейс и назначить его объекту *so_alt_channel_t::client_factory_t* в конструкторе. Например, так:

```
1  class my_logger_t
2    : public so_alt_channel_1::logger_t
3    {
4    private :
5      const so_4::rt::agent_t & m_owner;
6    public :
7      my_logger_t( const so_4::rt::agent_t & owner )
8        : m_owner( owner )
9        {}
10
11     virtual void
12     on_connect_attempt(
13       const std::string & alias,
14       size_t index,
15       const so_4::ret_code_t & rc )
16       {
17         if( rc )
18           so_log_1::err[ m_owner ]
19             [ so_log_1::n() << "Ошибка установления соединения" ]
20             [ so_log_1::d() << "alias: " << alias
21               << ", index: " << index
22               << ", error: " << rc ]( __FILE__, __LINE__ );
23       }
```

```
24
25    virtual void
26    on_switch_attempt(
27      const std::string & alias,
28      size_t index,
29      const so_4::ret_code_t & rc )
30      {
31        if( rc )
32          so_log_1::err[ m_owner ]
33            [ so_log_1::n() << "Ошибка переключения соединения" ]
34            [ so_log_1::d() << "alias: " << alias
35              << ", index: " << index
36              << ", error: " << rc ]( __FILE__, __LINE__ );
37      }
38  };
39
40 so_alt_channel_1::client_factory_t factory( factories, "default",
41   a_owner->query_name(),
42   // Назначаем логгера событий so_alt_channel.
43   new my_logger_t( *a_owner ) );
```

8.4 Пример использования

В качестве примера использования *so_alt_channel* используется код фабрики кооперaций для исходящих каналов из *so_sysconf* (6.2 на стр. 43):

```
1 /*
2   SoSysconf 2
3 */
4
5 /*!
6   \since v.2.3.0
7   \file
8   \brief Фабрика кооперaций 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_cln_channel.hpp>
18 #include <so_4/disp/active_group/h/pub.hpp>
19
20 #include <so_4/socket/channels/h/channels.hpp>
21
22 #include <so_4/sop/h/filter.hpp>
23
24 #include <so_alt_channel_1/h/pub.hpp>
25 #include <so_alt_channel_1/h/a_switch_enforcer.hpp>
26
27 #include <so_sysconf_2/h/coop_factory.hpp>
```

```
28 #include <so_sysconf_2/h/app_paths.hpp>
29 #include <so_sysconf_2/h/a_trouble.hpp>
30
31 #include <so_sysconf_2_ochannel/h/cfg.hpp>
32
33 namespace so_sysconf_2
34 {
35
36 namespace ochannel
37 {
38
39 //
40 // coop_factory_t
41 //
42
43 /*!
44   \since v.2.3.0
45   \brief Фабрика кооперaций ochannel-ов.
46
47   \par Псевдоним DLL
48   so_sysconf_2::ochannel
49
50   \par Имя фабрики
51   so_sysconf_2::ochannel::factory
52 */
53 class coop_factory_t
54   : public so_sysconf_2::coop_factory_t
55   {
56     //! Псевдоним для базового типа.
57     typedef so_sysconf_2::coop_factory_t  base_type_t;
58   public :
59     coop_factory_t()
60       : base_type_t(
61           "so_sysconf_2::ochannel",
62           "so_sysconf_2::ochannel::factory" )
63       {}
64
65     //! Регистрация новой кооперации.
66     /*!
67       Невозможность регистрации кооперации является
68       фатальной ошибкой.
69     */
70     virtual bool
71     reg(
72       const std::string & coop_name,
73       const std::string & cfg_file,
74       std::string & error_msg );
75   };
76
77 //! Предикат для std::for_each.
78 /*!
79   Добавляет в SOP-фильтр имя очередного агента.
80 */
81 class filter_filler_t
```

```
82    : public std::unary_function< const std::string &, void >
83    {
84      so_4::sop::std_filter_t & m_filter;
85    public :
86      filter_filler_t(
87        so_4::sop::std_filter_t & filter )
88        : m_filter( filter )
89        {}
90
91      result_type
92      operator()( argument_type a )
93        {
94          m_filter.insert( a );
95        }
96    };
97
98  //! Создать кооперацию агентов.
99  std::auto_ptr< so_4::rt::dyn_agent_coop_t >
100 make_coop(
101   //! Имя, которое должна иметь кооперация.
102   const std::string & coop_name,
103   //! Конфигурация кооперации.
104   const cfg_t & cfg )
105   {
106     // Имена агентов, которые будут входить в кооперацию.
107     std::string switch_enforcer_agent_name =
108         coop_name + "::a_switch_enforcer";
109     std::string channel_agent_name =
110         ( cfg.channel_agent_name().empty() ?
111             // Т.к. имя в конфиге не задано, то создаем его сами.
112             coop_name + "::a_channel" :
113             cfg.channel_agent_name() );
114
115     // Создаем фабрики для работы по альтернативным каналам.
116     std::auto_ptr< so_alt_channel_1::factories_t >
117         factories( new so_alt_channel_1::factories_t() );
118     std::transform(
119         cfg.addresses().begin(), cfg.addresses().end(),
120         std::back_inserter( *factories ),
121         so_4::socket::channels::create_client_factory );
122     std::auto_ptr< so_alt_channel_1::client_factory_t >
123         client_factory( new so_alt_channel_1::client_factory_t(
124             factories.release(), "default",
125             switch_enforcer_agent_name ) );
126
127     // Создаем агентов кооперации.
128     // Все агенты входят в одну активную группу.
129     std::auto_ptr< so_4::rt::agent_t >  a_switch_enforcer(
130         new so_alt_channel_1::a_switch_enforcer_t(
131             switch_enforcer_agent_name,
132             *client_factory,
133             channel_agent_name,
134             cfg.try_switch_timeout() * 1000 ) );
135     so_4::disp::active_group::make_member(
```

```
136         *a_switch_enforcer,
137         coop_name );
138 
139     // Для канала должен быть подготовлен специальный фильтр.
140     std::auto_ptr< so_4::sop::std_filter_t > filter(
141         so_4::sop::create_std_filter() );
142     std::for_each( cfg.filter_agent_list().begin(),
143         cfg.filter_agent_list().end(),
144         filter_filler_t( *filter ) );
145 
146     std::auto_ptr< so_4::rt::comm::a_cln_channel_t > a_channel(
147         new so_4::rt::comm::a_cln_channel_t(
148             channel_agent_name,
149             so_alt_channel_1::proxy_t::create( *client_factory ),
150             filter.release(),
151             so_4::rt::comm::a_cln_channel_base_t::
152                 create_def_disconnect_handler(
153                     cfg.reconnect_timeout() * 1000,
154                     cfg.restore_timeout() * 1000 ) ) );
155     so_4::disp::active_group::make_member(
156         *a_channel,
157         coop_name );
158 
159     // Создаем кооперацию и заботимся о том, чтобы кооперация
160     // уничтожила фабрику альтернативных каналов.
161     so_4::rt::agent_t * coop_agents[] =
162         {
163           a_switch_enforcer.release(),
164           a_channel.release()
165         };
166 
167     std::auto_ptr< so_4::rt::dyn_agent_coop_t > coop(
168         new so_4::rt::dyn_agent_coop_t(
169             coop_name, coop_agents,
170             sizeof( coop_agents ) / sizeof( coop_agents[ 0 ] ) ) );
171     so_4::rt::dyn_coop_controlled(
172         *coop,
173         client_factory.release() );
174 
175     return coop;
176   }
177 
178 bool
179 coop_factory_t::reg(
180   const std::string & coop_name,
181   const std::string & cfg_file,
182   std::string & error_msg )
183   {
184     cfg_t cfg;
185     if( load_cfg_file(
186         so_sysconf_2::app_paths_t::instance()->etc_file_name(
187             cfg_file ),
188         cfg, error_msg ) )
189       {
```

```
190         so_4::rt::dyn_agent_coop_helper_t helper(
191             make_coop( coop_name, cfg ).release() );
192         if( !helper.result() )
193           // Кооперация успешно создана.
194           return true;
195         else
196           error_msg = "unable to register coop" +
197               cpp_util_2::slexcast( helper.result() );
198       }
199 
200     so_sysconf_2::a_trouble_t::send_msg_fatal_error(
201       "so_sysconf_2_ochannel::coop_factory_t::reg()",
202       "unable_to_register_coop",
203       error_msg );
204 
205     return false;
206   }
207 
208 //! Этот объект и будет фабрикой.
209 coop_factory_t  g_factory;
210 
211 } /* namespace ochannel */
212 
213 } /* namespace so_sysconf_2 */
```

Созданием кооперации, в которую входит транспортный и управляющий агенты, занимается функция *make_coop* (строки 100–176). В строках 115–121 создаются обычные фабрики для всех IP-адресов из конфигурационного файла. А затем, в строках 122–125 создается специальная фабрика из *so_alt_channel*. Этой фабрике сообщается имя управляющего агента, который владеет сообщениями `msg_switch_required` и `msg_channel_switched`.

Строки 127–134 отвечают за создание управляющего агента, в качестве которого используется готовый агент `a_switch_enforcer_t` (см. 8.2.2 на стр. 54) из *so_alt_channel*.

В строках 139–144 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 77–96.

В строке 146 начинается создание транспортного агента, которому в качестве фабрики коммуникационных объектов передается экземпляр proxy (см. описание причины в 8.3.1 на стр. 55) — строка 149. А так же назначается disconnect_handler, который будет отвечать за восстановление соединения в случае разрывов связи, — строки 151–154.

Часть VI

Message Box API

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