Особенности корпоративных приложений

В идеальном мире информационная система организации состоит из единственного приложения, выполняющего все необходимые операции. В действительности даже самые маленькие компании используют несколько приложений для получения необходимой функциональности. Это обусловлено следующими причинами.

- Организации приобретают программные пакеты, созданные сторонними разработчиками.
- Между приложениями, созданными в разное время, существуют технологические отличия.
- Приложения, созданные разными разработчиками, имеют архитектурные отличия.
- Своевременный выпуск приложения на рынок важнее наличия в приложении встроенных средств интеграции.

В результате каждая организация вынуждена решать задачу обмена данными между приложениями, созданными с помощью различных языков программирования, предназначенных для выполнения на разных программных платформах и реализующих разные подходы к управлению бизнес-процессами.

Взаимодействуя с программными системами компании, пользователи (клиенты, бизнес-партнеры и сотрудники компании), как правило, не задумываются о том, каким образом осуществляется взаимодействие между всеми этими приложениями и подсистемами. В то же время выполнение каждой бизнес-функции может затрагивать сразу несколько внутренних систем компании. К примеру, клиент хочет изменить информацию о своем адресе, а также проверить, был ли получен его последний платеж. В типичной компании обработка подобного запроса может возлагаться на две системы: обслуживания клиентов и биллинга. Подобным образом размещение клиентом нового заказа требует координации целого ряда различных систем. Компания должна проверить идентификатор клиента, убедиться в его положительной кредитной репутации, проверить доступность необходимого товара на складе, выполнить заказ, рассчитать стоимость доставки, сформировать и отправить счет и т.д. Таким образом, размещение нового заказа затрагивает как минимум 5-6 различных систем компании. А с точки зрения клиента, размещение заказа -- это всего лишь одна бизнес-транзакция.

Для поддержания общих бизнес-процессов, а также совместного использования данных несколькими приложениями последние необходимо интегрировать друг с другом. Основной целью интеграции является обеспечение эффективного, надежного и безопасного обмена данными между интегрируемыми приложениями.

Как правило, разработчики интеграционных решений сталкиваются со следующими вызовами.

- **Ненадежность сети передачи данных**. Все интеграционные решения предполагают передачу информации между устройствами. В отличие от процессов, выполняющихся в пределах одного компьютера, распределенной вычислительной среде присущ целый ряд недостатков. Зачастую общающиеся системы разделены континентами, что вынуждает передавать данные по

телефонным линиям, сегментам локальных сетей, через маршрутизаторы, коммутаторы, общедоступные сети и спутниковые каналы связи. Доставка информации на каждом из этих этапов связана с задержкой и риском потери.

- **Низкая скорость передачи данных**. Время доставки данных через компьютерную сеть на порядок больше времени вызова локального метода. Таким образом, создание распределенного приложения требует применения иных принципов проектирования, чем создание приложения, выполняющегося в пределах одного компьютера.
- **Различия между приложениями**. Интеграционное решение должно учитывать все различия (язык программирования, платформа, формат данных), существующие между объединяемыми системами.
- **Неизбежность изменений**. Интеграционное решение должно иметь возможность адаптации к изменению объединяемых им приложений. Зачастую преобразования в одной системе влекут за собой непредсказуемые последствия для других систем. Поэтому при интеграции приложений важно уменьшить их взаимозависимость за счет так называемого слабого связывания.

Для преодоления описанных трудностей можно воспользоваться четырьмя основными подходами.

Взаимодействие через обмен файлами

Файлы -- это универсальный механизм хранения данных, встроенный в любую операционную систему и поддерживающийся любым языком программирования. Таким образом, один из самых простых способов интеграции приложений может быть основан на использовании файлов. Одно приложение создает файл, а другое приложение считывает его.

Наиболее существенное преимущество файлов заключается в том, что разработчики интеграционного решения не нуждаются в дополнительных сведениях о внутренней реализации интегрируемых приложений. Основная задача организации взаимодействия заключается в преобразовании форматов файлов (если это необходимо). Результатом подобного подхода является слабое связывание интегрируемых приложений. Единственными общедоступными интерфейсами приложений являются создаваемые этими приложениями файлы.

Простота реализации передачи файла обусловливается также отсутствием необходимости в привлечении дополнительных средств или пакетов для интеграции. Вместе с тем это приводит к возрастанию нагрузки, ложащейся на плечи разработчиков интеграционного решения. Объединяемые приложения должны использовать общее соглашение о формате, именовании и расположении файлов.

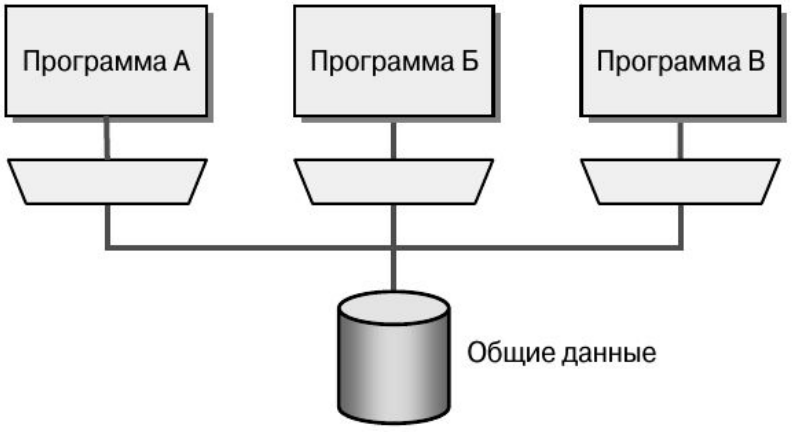
Приложение, создающее файл, должно обеспечить уникальность его имени. Также следует определить периодичность создания и считывания файлов приложениями. Слишком частая работа с файлами может привести к нерациональному использованию ресурсов.

Кроме того, необходимо разработать механизм удаления старых файлов, а также механизм блокировки доступа к файлу на время его записи. Если интегрируемые приложения не имеют доступа к общему диску, следует также обеспечить механизм перенесения файлов между дисками.

Один из наиболее существенных недостатков передачи файла заключается в возможности рассинхронизации интегрируемых систем вследствие низкой частоты обмена информацией. Если заказчик изменит адрес доставки товара утром, а система обслуживания клиентов распространит эту информацию ночью, то на протяжении всего дня система доставки товаров будет иметь неверные сведения о месте назначения груза. Впрочем, иногда рассинхронизция допустима. В конечном итоге люди уже привыкли к тому, что на распространение информации уходит какое-то время. Однако в некоторых случаях использование устаревшей информации может привести к крайне нежелательным последствиям. Таким образом, при определении частоты создания файлов следует всегда исходить из потребностей в наличии актуальной информации считывающих эти файлы приложений.

Взаимодействие через общую базу данных

Несколько приложений могут использовать общую логическую структуру данных, которой соответствует одна физическая база данных. Наличие единого хранилища данных устраняет проблему передачи информации между приложениями.



Одной из наибольших трудностей, присущих рассматриваемому стилю интеграции, является разработка схемы общей базы данных. Создание унифицированной схемы данных, удовлетворяющей требованиям нескольких различных приложений, связано со сложностями как технического, так и политического характера. Если применение унифицированной схемы данных приведет к снижению производительности критически важного приложения, руководство компании может настоять на необходимости пересмотра интеграционного проекта.

Если интегрируемые приложения расположены в нескольких географических точках, доступ к центральной базе данных будет осуществляться по традиционно медленным линиям связи глобальных сетей.

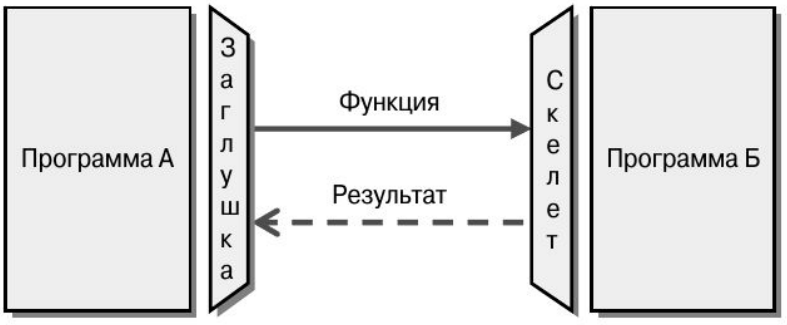
Еще одним камнем преткновения для реализации общей базы данных является коммерческое ПО. Большинство коммерческих приложений работает только со встроенной схемой данных, возможность адаптации которой зачастую оставляет желать лучшего.

Взаимодействие через удалённые вызовы процедур (RPC)

Одним из наиболее мощных механизмов структурирования приложений является инкапсуляция, предусматривающая доступ к данным посредством манипулирующего этими данными программного кода. Общая база данных представляет собой неинкапсулированную структуру, в то время как передача файла отличается замедленной реакцией на обновление данных.

Неинкапсулированный характер общей базы данных затрудняет поддержку интегрированных с ее помощью приложений. Изменения в приложении могут повлечь за собой изменения в базе данных, что, в свою очередь, скажется на всех оставшихся приложениях. Как следствие, организации, использующие общую базу данных, крайне неохотно относятся к необходимости ее изменения, что может затруднить реализацию новых бизнес-требований.

Выходом из сложившейся ситуации могло бы стать создание механизма, позволяющего приложению вызывать функцию другого приложения и передавать ей для обработки необходимые данные.



По сути, удаленный вызов процедуры представляет собой применение принципов инкапсуляции данных к интеграции приложений. Если приложение нуждается в получении или изменении информации, поддерживаемой другим приложением, оно обращается к нему посредством вызова функции. Таким образом, каждое приложение самостоятельно обеспечивает целостность своих данных и может изменять их формат, не затрагивая при этом оставшиеся приложения.

Следует отметить, что удаленный вызов процедуры характеризуется достаточно сильной степенью связывания приложений.

Взаимодействие через обмен сообщениями

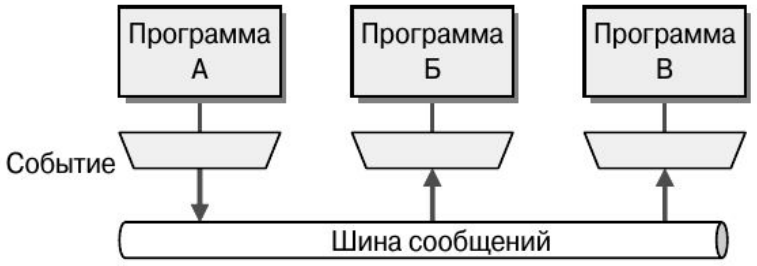
Передача файла и общая база данных позволяют приложениям получить доступ к общим данным, но не к общей функциональности. Удаленный вызов процедуры устраняет этот недостаток за счет сильного связывания интегрируемых приложений.

Зачастую же задача интеграции заключается в обеспечении своевременного обмена данными между слабо связанными приложениями.

Асинхронный обмен сообщениями устраняет большинство недостатков распределенных систем. Для передачи сообщения не требуется одновременной доступности отправителя и получателя. Более того, сам факт асинхронного обмена данными побуждает разработчиков к созданию компонентов, не требующих частого удаленного взаимодействия.

Подобно передаче файла система обмена сообщениями обеспечивает слабое связывание объединяемых приложений. Сообщения могут быть преобразованы во время передачи без уведомления отправителя или получателя. Слабое связывание позволяет разработчикам использовать различные способы доставки сообщений: широковещательную рассылку всем получателям, маршрутизацию сообщений одному получателю или их группе и т.п.

Необходимость преобразования данных обусловлена наличием у приложений различных концептуальных моделей, т.е. семантическим диссонансом. В отличие от общей базы данных, обмен сообщениями не предполагает использования специальных средств для его устранения. Это связано с тем, что семантический диссонанс неизбежно возникает при добавлении к интеграционному решению новых приложений (например, в результате слияния информационных систем компаний).



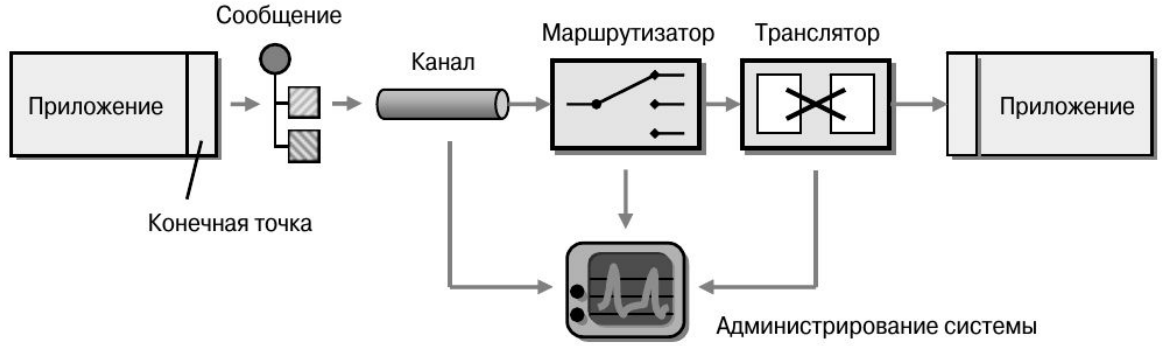
Частый обмен небольшими порциями данных создает предпосылки для использования приложениями общей функциональности. Если получение сообщения о размещении нового заказа требует выполнения некоторых действий, они могут быть инициированы путем отправки специальных сообщений. Несмотря на то что скорость подобного взаимодействия ниже, чем при использовании удаленного вызова процедуры, вызывающей стороне не приходится ждать ответа на отправленное сообщение. Но на самом деле обмен сообщениями не такой уж и медленный -- достаточно большое число систем обмена сообщениями используются финансовыми организациями для обработки тысяч котировок акций в секунду.

Message-oriented middleware

Рассмотрим типичные компоненты связующего ПО, ориентированного на обмен сообщениями (message-oriented middleware, MOM).

Примером данных, передаваемых между приложениями, может являться адрес заказчика, вызов удаленной службы или фрагмент HTML-кода информационного портала. Передачу данных между приложениями обеспечивают два компонента интеграционного решения: *сообщение* и *канал*. Коммуникационный канал предназначен для обмена информацией между приложениями. В качестве канала

может использоваться TCP/IP-соединение, очередь сообщений, общий файл, общая база данных и даже USB-флешка. В канал помещается сообщение -- фрагмент данных, который имеет одинаковое значение для обоих интегрируемых приложений. Объем информации, передаваемый с помощью одного сообщения, может быть как очень маленьким (например, телефонный номер заказчика), так и довольно большим (например, список всех заказчиков и адресов их проживания).



Несмотря на то что передача сообщений по каналам уже может считаться ‘‘простой’’ формой интеграционного решения, не стоит останавливаться на достигнутом. Как правило, разработчики интеграционного решения не могут изменять объединяемые приложения, в частности их внутренний формат данных. К примеру, одна из интегрируемых систем может хранить имя клиента с помощью полей FIRST_NAME и LAST_NAME, а другая -- с помощью единственного поля Customer_Name. Поскольку возможность изменить внутренний формат данных приложений выпадает крайне редко, связующее ПО обязано обеспечить механизм преобразования форматов данных между приложениями. Это осуществляется компонентами, называемыми *трансляторами сообщений*.

Предположим, что нам необходимо интегрировать более двух систем. Как изменится механизм обмена данными в этом случае? Наиболее простое решение состоит в указании системой-отправителем адресов систем-получателей сообщения. К примеру, при изменении адреса клиента система обслуживания заказчиков пересылает обновленную информацию всем системам, хранящим адрес клиента. Данный подход имеет один существенный недостаток: каждая система-отправитель должна поддерживать информацию обо всех системах-получателях сообщения. Как следствие добавление новой системы может потребовать внесения изменений в существующие системы интеграционного решения. Было бы гораздо лучше, если бы функция доставки сообщений получателям была возложена на связующее ПО, а именно на его *маршрутизирующий* компонент, например, такой как брокер сообщений.

Как правило, со временем все интеграционные решения становятся слишком сложными. Приложения, форматы данных, каналы, маршрутизаторы, преобразователи -- все эти элементы зачастую распределены между множеством операционных платформ и географических размещений. С целью управления интеграционным решением связующее ПО должно включать в себя подсистему *администрирования*. Основные задачи подсистемы администрирования состоят в отслеживании потоков данных, работоспособности компонентов и приложений, а также уведомлении о возникших ошибках.

Казалось бы, мы учли все, что необходимо для создания полноценного интеграционного решения, -- компоненты передачи данных, преобразования формата, маршрутизации сообщений и администрирования. Однако мы исходили из допущения, что интегрируемые приложения могут самостоятельно помещать сообщения в канал. К сожалению, большинство legacy-приложений, коммерческих приложений, а также приложений, созданных на заказ, изначально не предназначались для интеграции. Для ‘‘подключения’’ подобных приложений к интеграционному решению используется *конечная точка* сообщения -- специализированный компонент или адаптер канала, предоставленный разработчиком интеграционной платформы.

Рассмотрим теперь эти компоненты и их вариации подробнее.

Каналы

Приложение, которому необходимо отправить информацию, помещает ее не просто в систему обмена сообщениями, а в конкретный канал сообщений. Подобным образом приложение, которому необходимо получить информацию, обращается не просто к системе обмена сообщениями, а к конкретному каналу сообщений.

В действительности каналы представляют собой логические адреса в системе обмена сообщениями. Реализация каналов сообщений зависит от конкретной системы обмена сообщениями. К примеру, все конечные точки сообщения могут быть соединены друг с другом или подключены к центральному коммутатору, а несколько логических каналов сообщений могут быть представлены одним физическим каналом. Как бы там ни было, логические каналы скрывают детали конкретной реализации от приложений.

Система обмена сообщениями не содержит заранее сконфигурированных каналов сообщений. Требуемые каналы сообщений определяются на этапе проектирования интеграционного решения и реализуются программистами, внедряющими систему обмена сообщениями. Несмотря на то, что некоторые системы обмена сообщениями поддерживают создание новых каналов сообщений после начала эксплуатации интеграционного решения, это не позволяет распространить информацию о новых каналах между всеми приложениями. Следовательно, количество и назначение каналов сообщений необходимо определить до этапа развертывания программной системы.

Выделяют несколько типовых вариантов каналов.

Канал “точка-точка”

Канал “точка-точка” гарантирует, что каждое отдельное сообщение будет потреблено только одним получателем. У канала может быть много получателей, одновременно обрабатывающих сообщения, но только один из них сможет успешно принять конкретное сообщение. Если несколько получателей попытаются одновременно потребить некоторое сообщение, канал сам позаботится о том, чтобы эта операция удалась только одному из них. Координировать свои действия получателям не придется.

Канал “публикация-подписка”

Канал “публикация-подписка” используется, когда сообщение должно быть обработано не одним, а всеми получателями, прослушивающими канал. Сообщение о событии не может считаться потребленным, пока уведомление не получат все подписчики. Когда же это произойдет, сообщение считается потребленным и должно быть удалено из канала. Канал ‘‘публикация-подписка’’ функционирует следующим образом: у него есть один входной канал, который разбивается на несколько выходных каналов, по одному на каждого подписчика. Когда оповещение о событии публикуется в канале, канал ‘‘публикация-подписка’’ доставляет копию сообщения в каждый из выходных каналов. На каждом ‘‘выходе’’ канала есть только один подписчик, которому разрешается потреблять сообщение только один раз. Благодаря этому каждый подписчик получит сообщение только единожды, после чего потребленные копии сообщения исчезнут из соответствующих выходных каналов.

Помимо всего прочего, канал ‘‘публикация-подписка’’ удобно применять для отладки. Даже если сообщение предназначено для одного получателя, использование такого канала позволяет прослушивать канал сообщений, не вмешиваясь в существующий поток сообщений. Он избавляет разработчика от необходимости вставлять в код приложений, участвующих в обмене сообщениями, множественные выражения отладочного вывода. Однако, использование данного канала должно быть ограничено политикой безопасности. В отличие от канала “точка-точка” отследить добавление нового потребителя сообщений в таком канале может быть довольно сложно.

Каналы по типам данных

В случае, когда приложения должны обмениваться сообщениями с разными типами данных, часто бывает удобно развести разные типы сообщений по разным каналам. Этого же можно добиться, добавив в заголовок сообщения какой-нибудь идентификатор типа, однако тогда на получающей стороне пришлось бы делать switch по этим типам. Да и к тому же наличие отдельного канала для каждого типа данных (или некоторых из них) позволяет осуществлять приоритезацию трафика (например, какие-то данные должны быть доставлены обязательно и в первую очередь, а другие не страшно и потерять) и эффективно перенаправлять определённые данные при необходимости (опять же, удобно на стадии отладки).

Канал некорректных сообщений

Канал некорректных сообщений представляет способ убрать неверные сообщения из канала и поместить их в такое место, где они не будут мешать потоку других сообщений, но могут быть найдены для диагностики проблем в системе обмена сообщениями. Проблемные сообщения могут быть некорректны синтаксически, семантически, находиться не в том канале или просто не соответствовать текущему контексту получателя. Игнорировать их нехорошо, помещать обратно в канал, откуда они пришли, бессмысленно (некорректное сообщение может попытаться обработать кто-то другой). Канал недопустимых сообщений -- это своеобразный журнал записи

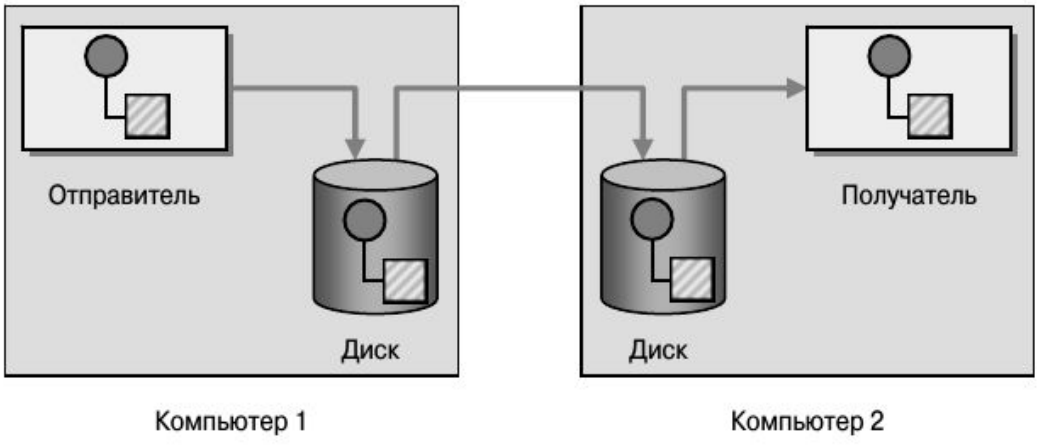
ошибок для системы обмена сообщениями, его получателем будет некий системный компонент обработки ошибок.

Канал недоставленных сообщений

Канал недоставленных сообщений, как несложно понять, используется для хранения сообщений, которые в силу определённых причин не смогли быть получены и обработаны. Причины могут быть разные -- может истечь срок действия сообщения, в заголовке сообщения может быть указан тип данных, в котором не заинтересован ни один из получателей, в заголовке в принципе могут быть какие-нибудь ошибки. Разница между недоставленным (dead) и недопустимым (invalid) сообщениями заключается в следующем: решение о том, следует ли отправить сообщение в канал недоставленных сообщений, принимается системой обмена сообщениями путем оценки заголовка. В отличие от этого отправку сообщения в канал недопустимых сообщений осуществляет получатель (из-за ошибок в теле сообщения, отсутствия интересующих его полей в заголовке и т.п.). Для получателя обнаружение и обработка недоставленных сообщений выглядит автоматизированным процессом, в то время как заниматься судьбой недопустимых сообщений должен он сам.

Гарантированная доставка

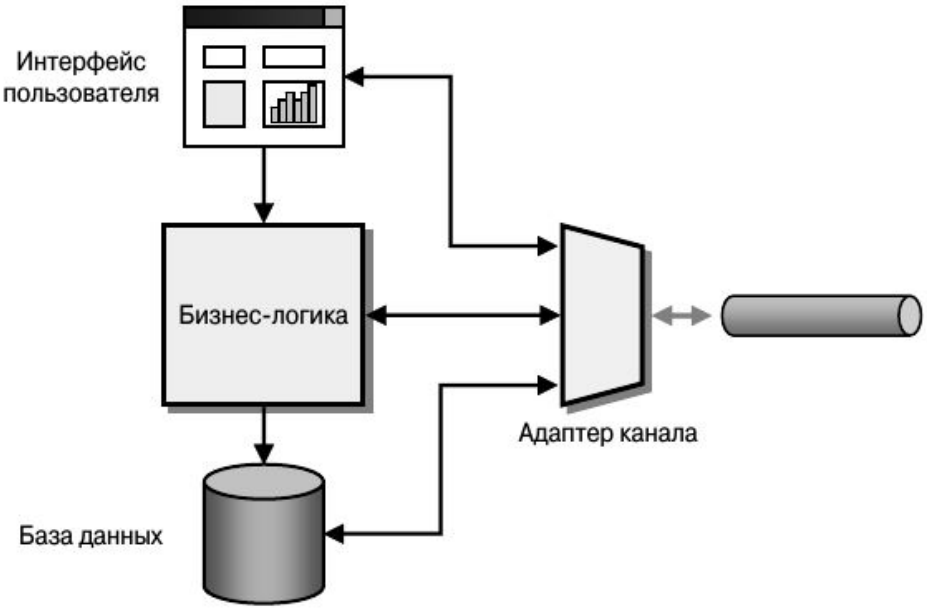
У каждого компьютера, на котором установлена система обмена сообщениями, имеется собственное хранилище данных, чтобы сообщения хранились локально. Когда отправитель отсылает сообщение, операция отправки не считается успешно завершенной до тех пор, пока сообщение не будет надежно сохранено в хранилище данных отправителя. Аналогичным образом сообщение не будет удаляться из указанного хранилища, пока не будет успешно передано и сохранено в следующем хранилище, и т.п. Итак, с момента отправки сообщение всегда будет храниться на диске как минимум одного компьютера до тех пор, пока не будет успешно доставлено получателю и принято им. Наличие постоянного хранилища увеличивает надежность, но приводит к снижению производительности. Не следует забывать и о том, что реализация гарантированной доставки в системах с высоким уровнем трафика может потребовать огромного количества дискового пространства. По этой причине в некоторых системах обмена сообщениями разрешается настраивать таймаут повторных попыток (retry timeout).



Гарантированную доставку также рекомендуется отключать на время тестирования и отладки. Тогда для полной очистки каналов будет достаточно остановить и перезапустить сервер обмена сообщениями. Сообщения, стоящие в очереди и после перезапуска сервера, могут сильно затруднить отладку даже простых программ для обмена сообщениями.

Адаптер канала

Адаптер канала позволяет организовать взаимодействие с подсистемой или приложением, которая создавалась без учёта системы обмена сообщениями. Адаптер канала осуществляет доступ к API или данным изолированного приложения, создаёт на основе этих данных сообщение и размещает его в канале. Аналогичным образом адаптер канала может получать сообщения, а затем в зависимости от их содержимого вызывать в приложении ту или иную функциональность.



Адаптер канала может подключаться к различным уровням архитектуры приложения. В зависимости от особенностей архитектуры и того, какие данные нужны системе обмена сообщениями, выделяют несколько типов адаптеров: адаптер пользовательского интерфейса (работает напрямую с UI, графическим или консольным), адаптер бизнес-логики (работает с предоставляемым приложением API) и адаптер базы данных (работает напрямую с данными приложения). Отметим тут, что в большинстве случаев адаптеры каналов приходится использовать совместно с трансляторами сообщений (см. про них далее).

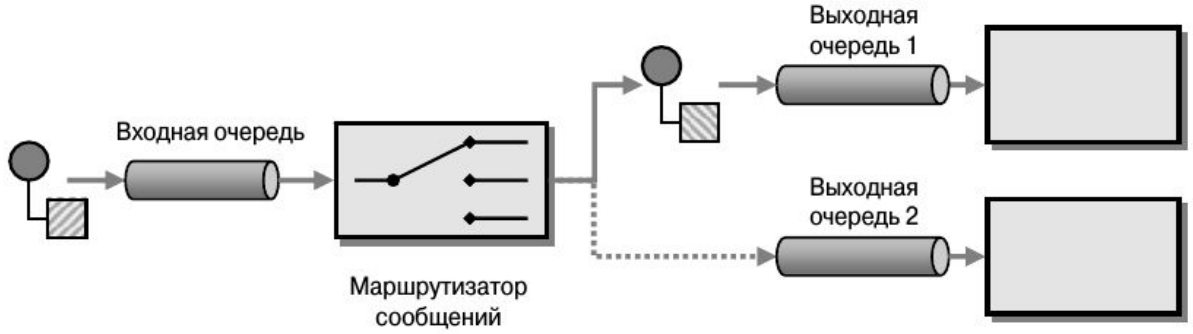
Особой разновидностью адаптера канала является мост обмена сообщениями (Messaging Bridge). Он соединяет систему обмена сообщениями не с конкретным приложением, как обыкновенный адаптер, а с другой системой обмена сообщениями.

Маршрутизаторы сообщений

Канал сообщений позволяет провести четкую границу между отправителем и получателем сообщения. В частности, из этого следует, что несколько приложений могут публиковать сообщения в одном и том же канале сообщений. В результате канал сообщений может содержать сообщения, способ обработки которых будет зависеть от их типа, источника или какого-либо другого критерия. Несмотря на то что для передачи сообщения каждого типа можно задействовать отдельный канал сообщений, это приведет к необходимости классификации сообщений отправителями, а также к существенному росту числа каналов сообщений. К тому же способ обработки сообщения не всегда зависит от его источника. Представим ситуацию, в которой получатель сообщения определяется динамически на основе общего числа сообщений, переданных по конкретному каналу. Поскольку это число не может знать ни один из отправителей, он не сможет выбрать корректный канал для размещения сообщения.

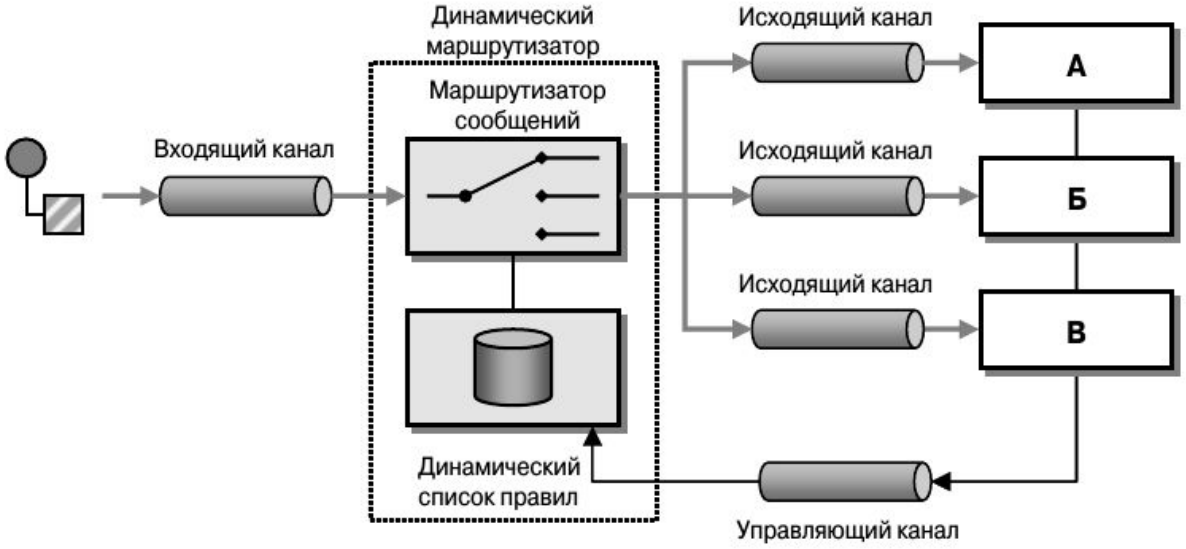
Канал сообщений обеспечивает простую форму маршрутизации сообщений. Публикуя сообщение в канале сообщений, приложение теряет контроль за его доставкой. Отныне маршрут сообщения зависит от компонента, находящегося на другом конце канала сообщений. Решение о необходимости обработки сообщения, поступившего по общему каналу сообщений, можно возложить на получателя. Однако как только сообщение будет извлечено из канала, оно перестанет быть доступным для проверки другим компонентам. Некоторые системы обмена сообщениями позволяют просмотреть свойства сообщения без его извлечения из канала, тем не менее это решение не является универсальным и к тому же ‘‘привязывает’’ компонент к определенному типу сообщения.

Использование маршрутизатора сообщений позволяет сосредоточить всю логику принятия решения о доставке сообщения в одном компоненте. Отныне определение новых типов сообщений, добавление новых этапов обработки или изменение правил маршрутизации затронет только один компонент, маршрутизатор сообщений. Кроме того, прохождение всех входящих сообщений через маршрутизатор сообщений делает возможной их обработку в корректном порядке. Ключевое свойство маршрутизатора сообщений состоит в том, что он не изменяет содержимого сообщения, заботясь только о его корректной доставке.



К сожалению, маршрутизатору сообщений свойственны определенные недостатки. В частности, он должен обладать информацией обо всех доступных каналах сообщений. Если эти сведения часто меняются, маршрутизатор сообщений

может превратиться в ‘‘узкое место’’ интеграционного решения. В этом случае рекомендуется переложить ответственность за выбор сообщения на получателя, воспользовавшись каналом ‘‘публикация-подписка’’ и набором фильтров сообщений, либо задав правила маршрутизации в виде внешнего динамического списка правил. Архитектура такого решения сильно усложняется, особенно если позволить получателям как-то влиять на правила маршрутизации, однако получается большая гибкость конечного решения.

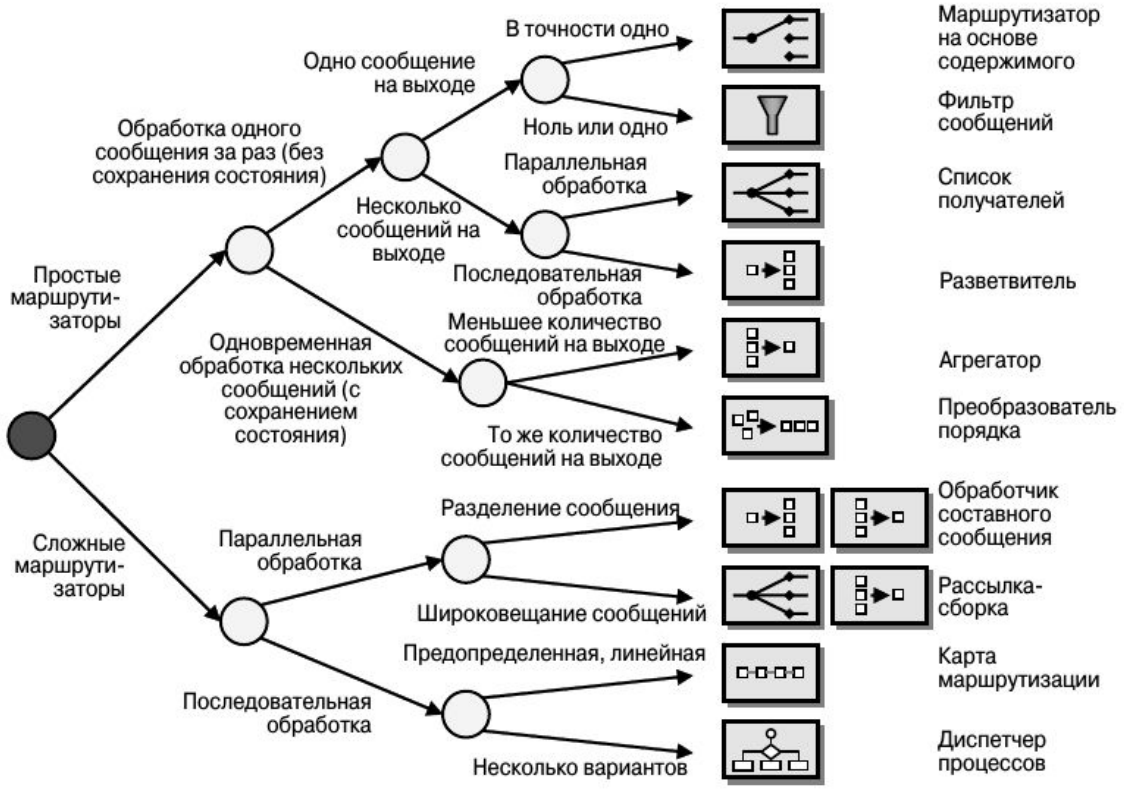


Самый простой вид маршрутизатора -- фиксированный маршрутизатор, он имеет один входящий и один исходящий канал. Функциональность фиксированного маршрутизатора -- извлечь сообщение из входящего канала и поместить его в исходящий канал. Зачем же нужен такой примитивный маршрутизатор? Во-первых, фиксированный маршрутизатор может использоваться как временное решение до реализации более интеллектуальной логики маршрутизации. Во-вторых, фиксированный маршрутизатор может применяться для перенаправления сообщений между несколькими интеграционными решениями. Зачастую вместе с фиксированным маршрутизатором используются транслятор сообщений и адаптер канала.

Маршрутизаторы на основе контекста принимают решение о точке назначения сообщения на основании условий среды. Как правило, маршрутизаторы на основе контекста применяются для балансировки нагрузки, тестирования или обеспечения восстановления при отказе. К примеру, если компонент выйдет из строя, маршрутизатор на основе контекста сможет перенаправить сообщения на обработку другим компонентом с аналогичной функциональностью. Некоторые маршрутизаторы на основе контекста разделяют поток сообщений между несколькими каналами для достижения эффекта балансировки нагрузки.

Многие маршрутизаторы сообщений являются маршрутизаторами без поддержки состояния. Другими словами, принятие решения о передаче сообщения принимается без учета ранее переданных сообщений. Маршрутизаторы, которые учитывают обработанные ранее сообщения при принятии решения о точке назначения текущего сообщения, называются маршрутизаторами с поддержкой состояния. Примером маршрутизатора с поддержкой состояния является компонент, реализующий защиту от дублирования сообщений.

Более конкретно можно выделить следующие основные типы маршрутизаторов:



Маршрутизаторы на основе содержимого

Маршрутизаторы на основе содержимого принимают решение о точке назначения сообщения на основании его свойств, например типа сообщения или значения определенных полей.

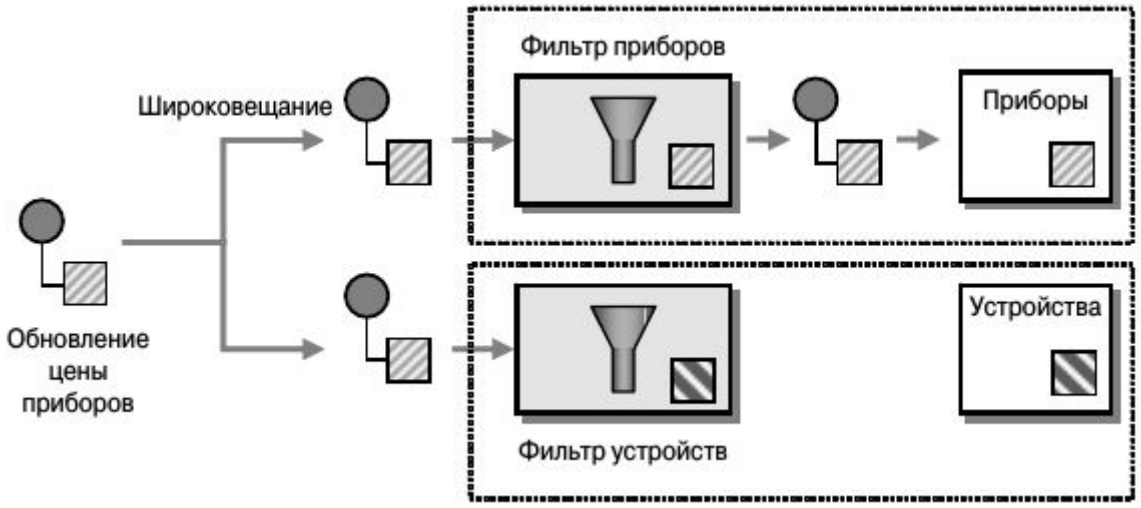
Фильтры сообщений

Фильтры сообщений позволяют компонентам не получать лишних сообщений. Вообще самый простой способ избавить себя от ненужных сообщений подписаться только на те каналы, по которым доставляется интересующая информация. Этот подход задействует встроенную способность к маршрутизации, которой обладают каналы ‘‘публикация-подписка’’. Компонент получает только сообщения, передающиеся по каналам, на которые он подписан. К примеру, можно создать один канал для рассылки обновлений цен на приборы и еще один канал, в котором будут публиковаться обновления цен на устройства. В этом случае клиенты компании смогут сами выбирать, на какой канал или каналы подписываться. У данного подхода есть еще одно преимущество: добавление новых подписчиков не потребует внесения изменений в систему. С другой стороны, получение сообщений по каналу ‘‘публикация-подписка’’ обычно ограничено простым бинарным условием: если компонент подписан на канал, он получает все сообщения из этого канала. Единственный способ предоставить компоненту больше выбора в том, какие сообщения получать, а какие нет, создать как можно больше каналов. При работе с комбинацией нескольких параметров число каналов очень быстро вырасти. К примеру,

если мы хотим рассылать сведения о скидках трёх видов на два разных товара, нам потребуется целых шесть каналов ‘‘публикация-подписка’’.

В нашем примере для реализации такого подхода следует создать один общий канал ‘‘публикация-подписка’’, который будет прослушиваться всеми клиентами. После этого клиент сможет воспользоваться фильтром сообщений, чтобы отбирать сообщения на основании собственных предпочтений, как, например, тип продукции или величина скидки.

Фильтр сообщений можно рассматривать как частный случай маршрутизатора на основе содержимого, направляющего полученное сообщение в исходящий канал либо в так называемый нулевой канал (null channel), который уничтожает все опубликованные в нем сообщения.



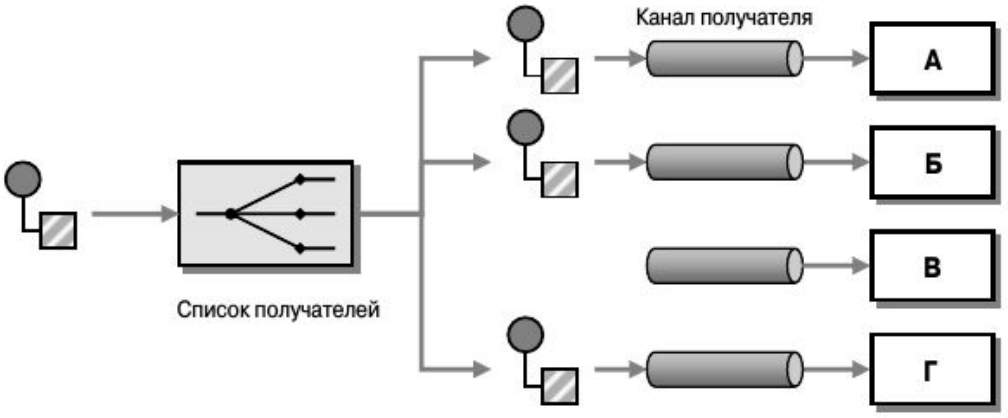
Иногда выбор того или иного решения определяется требуемой функциональностью. В большинстве случаев выбор решения зависит от того, какие компоненты системы должны управлять маршрутизацией сообщений. Контроль над маршрутизацией должен быть централизованным, или же его лучше переложить на плечи получателей? Если сообщения содержат секретные данные, которые должны быть доступны только конкретным получателям, необходимо использовать маршрутизатор на основе содержимого -- опасно доверять фильтрацию сообщений другим получателям.

На выбор варианта реализации могут повлиять и соображения, связанные с работой сети. Если у вас есть удобный способ широковещательной рассылки информации (например, многоадресная IP-рассылка по внутренней сети), использование фильтров может оказаться весьма эффективным, а заодно избавит систему от потенциального ‘‘узкого места’’ в виде единственного маршрутизатора. Но если информацию приходится передавать по Интернету, вы будете ограничены соединениями типа ‘‘точка-точка’’. В подобных случаях предпочтительнее использовать централизованный маршрутизатор, поскольку он позволит избежать отправки сообщений всем получателям вне зависимости от интересов последних.

Список получателей

Маршрутизация на основе списка получателей позволяет отправителям указывать желаемых получателей. Список может динамически меняться, что даёт

преимущество перед обычным каналом “публикация-подписка”. Этот подход также не заставляет рассылать много потенциально лишних сообщений, как при использовании фильтров.



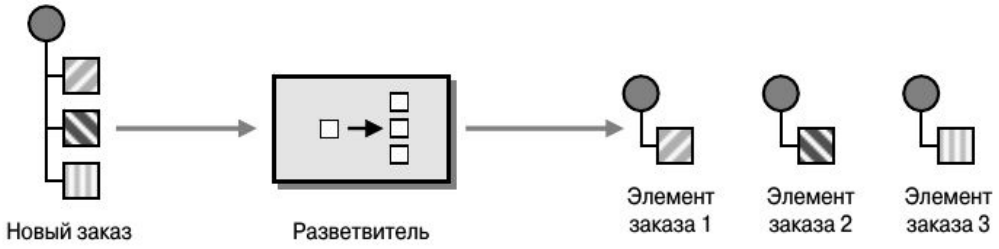
Подобный маршрутизатор может составлять список получателей также и самостоятельно, исходя из содержимого сообщения и внутреннего набора правил.

Динамический список получателей может применяться для реализации канала ‘‘публикация-подписка’’, если система обмена сообщениями обладает только каналами ‘‘точка-точка’’. В этом случае в списке получателей будет храниться перечень всех каналов ‘‘точка-точка’’, получатели которых подписаны на определенную тему. Каждая тема может быть представлена одним конкретным экземпляром списка получателей. Это решение может пригодиться и тогда, когда необходимо применить специальные критерии, чтобы дать получателям возможность подписаться на некоторый источник данных. В отличие от каналов ‘‘публикация-подписка’’, на которые может подписаться любой компонент, в список получателей легко внедрить логику, ограничивающую доступ к источнику данных путем внесения в список лишь избранных подписчиков. При этом, конечно же, предполагается, что подписчики не могут получать сообщения непосредственно из канала, ведущего к списку получателей.

Разветвитель

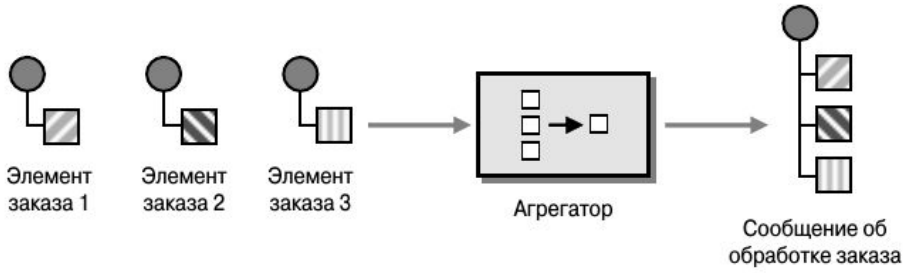
Разветвитель позволяет разбить сложное сообщение на несколько простых. Каждое из них будет содержать данные, касающиеся определённого аспекта оригинального сообщения.

Для каждого элемента (или подмножества элементов) входящего сообщения разветвитель публикует по одному отдельному сообщению. Во многих случаях желательно, чтобы в каждом из полученных сообщений повторялись некоторые общие элементы данных. Это необходимо для того, чтобы каждое дочернее сообщение стало самодостаточным и могло обрабатываться без сохранения состояния. Кроме того, наличие таких избыточных элементов впоследствии позволит скомпоновать дочерние сообщения в одно. К примеру, в каждом сообщении с информацией об элементе заказа должен содержаться номер заказа, чтобы элемент заказа впоследствии можно было снова сопоставить с заказом и другими связанными с ним сущностями, например с клиентом, разместившим заказ.



Агрегатор

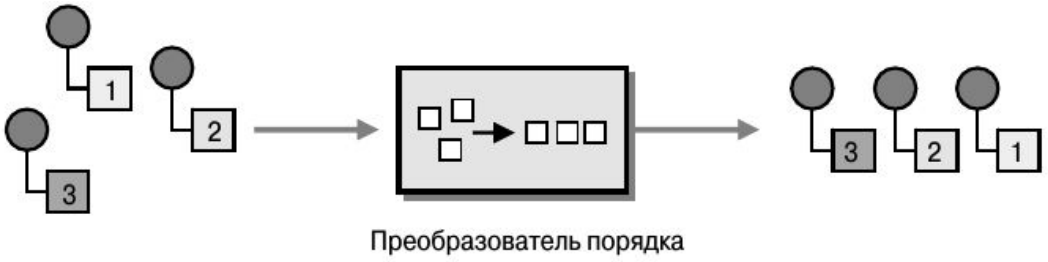
Агрегатор выполняет задачу, обратную разветвителю. Это фильтр с сохранением состояния, который собирает и сохраняет отдельные сообщения, пока не получит полный набор. После этого агрегатор составляет на их основе единое сообщение и публикует его для дальнейшей обработки.



Агрегатору не обязательно хранить каждое входящее сообщение целиком. К примеру, если речь идет о предложениях на аукционе, достаточно хранить лишь предложение с самой лучшей на текущий момент ценой и соответствующий идентификатор участника торгов. Несмотря на это агрегатор все равно должен хранить некоторую информацию в промежутке между поступлением сообщений, значит, он работает с сохранением состояния.

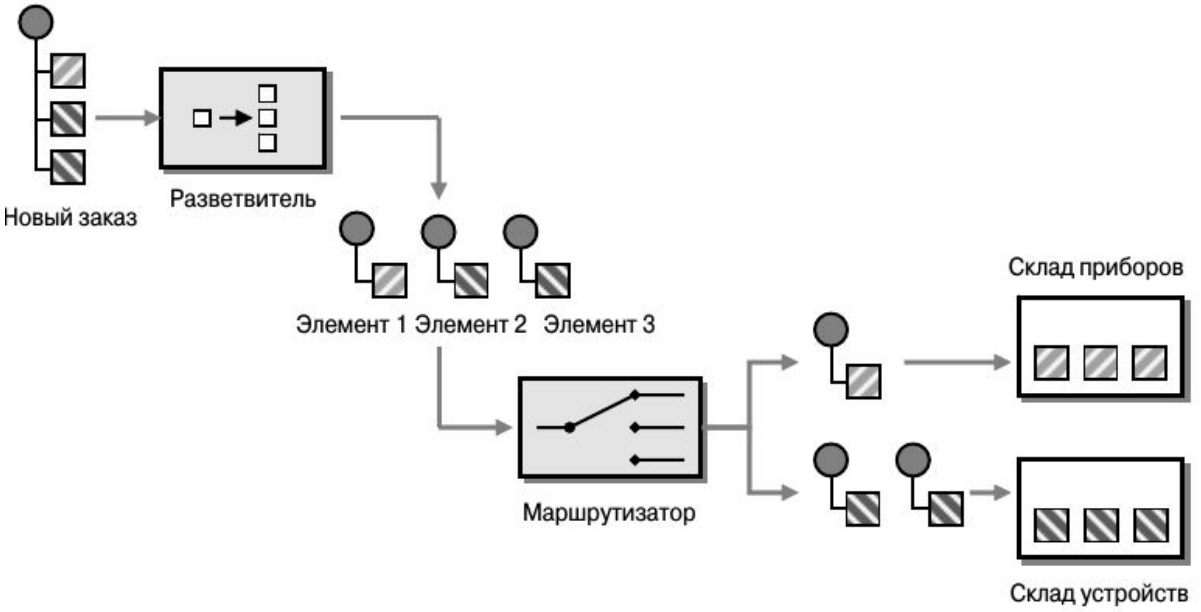
Преобразователь порядка

Преобразователь порядка получает поток сообщений с потенциально нарушенной очередностью, сохраняет их во внутреннем буфере до тех пор, пока не получит всю последовательность целиком, а затем публикует сообщения в исходящем канале в правильном порядке. Как и большинство других маршрутизаторов, преобразователь порядка обычно не вносит изменений в содержимое сообщения. Как и агрегатор, это тоже фильтр с сохранением состояния.



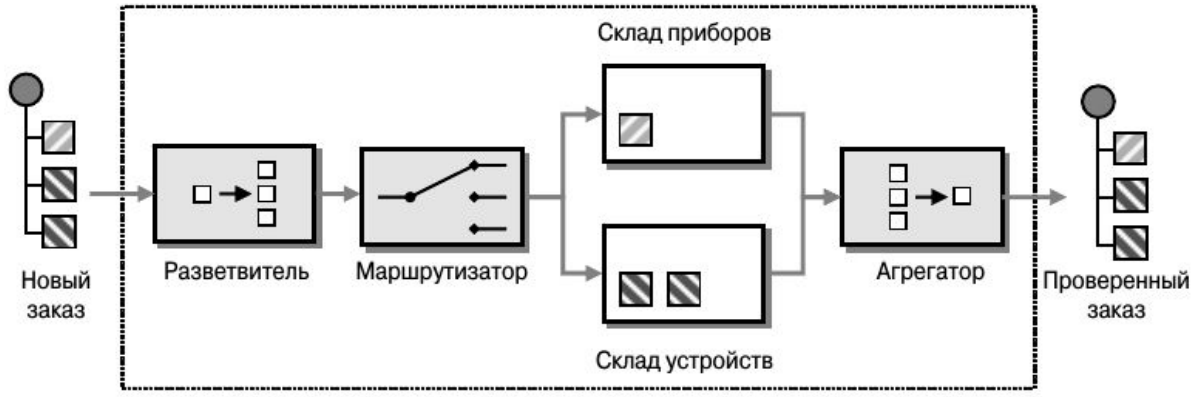
Обработчик составного состояния

Данный шаблон позволяет провести по маршруту его следования сообщение, которое состоит из нескольких элементов, каждый из которых проходит собственную обработку. По идее, это задачу можно было бы решить с помощью разветвителя и маршрутизатора по данным:



В нашем примере это означает, что каждый элемент заказа направляется на проверку в соответствующую систему управления складскими запасами. Указанные системы полностью отделены друг от друга, и каждая из них получает только те элементы заказа, которые соответствуют хранящимся на ее складах видам товара. Но приведённое решение не позволяет определить, все ли элементы заказа действительно найдены на складах и готовятся к отправке. Кроме того, хотелось бы, чтобы дальнейшая обработка заказа происходила так, как если бы он все еще был одним сообщением, даже несмотря на то, что его уже разбили на несколько более мелких сообщений.

Один из возможных подходов состоит в том, чтобы просто собрать все элементы заказа, прошедшие через конкретную систему управления складскими запасами, и сформировать на их основе отдельный заказ. После этого такой заказ может обрабатываться, как единое целое: он будет собран и доставлен, после чего клиенту будет выслан счет. Каждый подзаказ будет рассматриваться как независимый процесс. В некоторых ситуациях из-за отсутствия централизованного контроля над дальнейшим процессом обработки такой подход может стать единственным допустимым решением.



Поскольку все сообщения, касающиеся отдельных элементов заказа, появляются на свет в результате разбивки одного и того же сообщения, агрегатору можно передать дополнительную информацию, а именно число производных сообщений, чтобы применить более эффективную стратегию агрегации. Несмотря на это обработчику составного сообщения приходится сталкиваться с рядом других проблем, связанных с пропавшими или задерживающимися сообщениями. Как следует поступить, если система управления складскими запасами временно недоступна? Нужно ли повторно отправлять сообщение с запросом при отсутствии одного из ответов?

Обработчик составного сообщения является хорошим примером слияния нескольких независимых шаблонов проектирования в более сложный шаблон. Для остальных компонентов системы обработчик составного сообщения будет выглядеть обычным фильтром с одним входящим и одним исходящим каналами, обеспечивая тем самым эффективную абстракцию сложного внутреннего устройства.

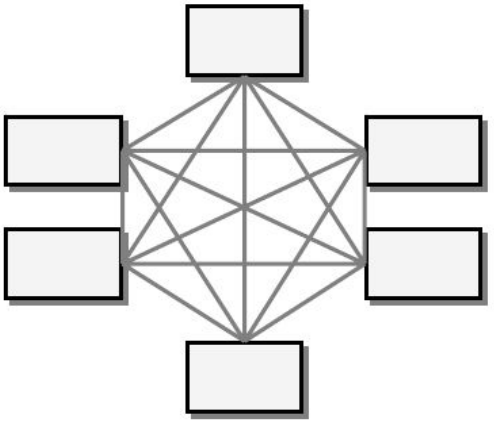
Брокер сообщений

Многие шаблоны проектирования, о которых рассказывалось выше, описывают способы маршрутизации сообщения в требуемую точку назначения без какого-либо участия со стороны приложения-отправителя. Большинство этих шаблонов посвящено конкретным типам логики маршрутизации. В совокупности, однако, они могут решить более глобальную проблему -- отделение пункта назначения сообщения от его отправителя, сохраняя централизованный контроль над сообщениями.

Использование простого канала сообщений само по себе уже обеспечивает наличие уровня косвенности между отправителем и получателем отправитель знает только о канале, но не о получателе. Однако, если у каждого получателя будет собственный канал, такой уровень косвенности во многом потеряет свою значимость. Вместо адреса получателя отправитель должен будет знать имя канала, соответствующего этому получателю.

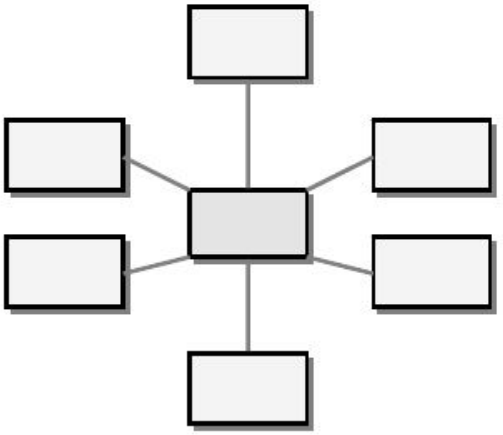
Интеграционные архитектуры часто появляются на свет в результате постепенного расширения некогда небольшой системы. Вначале системе обслуживания клиентов нужно было общаться только с системой бухгалтерского учета. Затем от системы обслуживания клиентов стали также требовать, чтобы она извлекала информацию из системы управления складскими запасами, а системе отгрузки товара поручили обновлять систему бухгалтерского учета, внося в нее

расходы по погрузке, и т.д. Как видим, очередное добавление в систему ‘‘еще одного элемента’’ может легко нарушить общую целостность решения.



Шаблоны маршрутизации сообщений помогают отделить отправителя от получателей. Однако при большом количестве взаимодействующих компонентов может появляться большое число маршрутизаторов, и ими тоже приходится управлять.

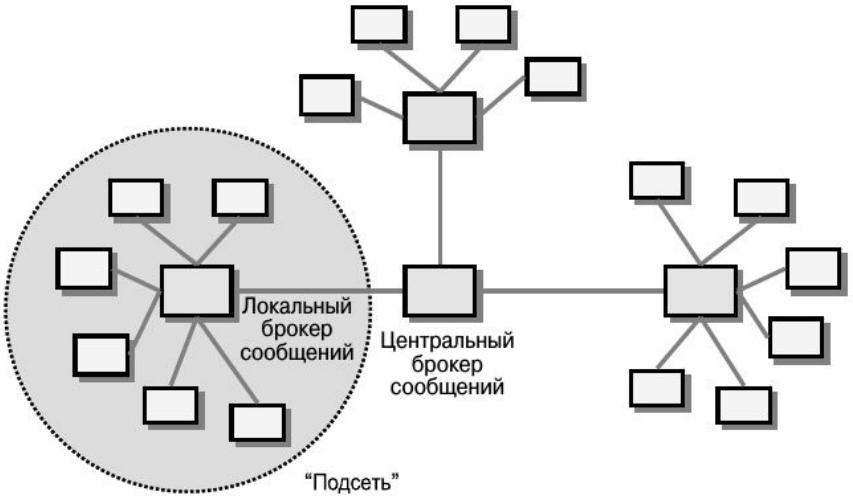
Использование централизованного брокера сообщений часто определяют как центрально-лучевую топологию, или архитектурный стиль ‘‘звезда’’.



Шаблон брокер сообщений имеет несколько иную направленность, чем большинство шаблонов маршрутизации, описанных ранее. В отличие от них он является не шаблоном проектирования, а архитектурным шаблоном, следовательно, его можно сравнить с архитектурным стилем “каналы и фильтры” (Pipes and Filters). Напомним, что последний описывает фундаментальный способ связывания отдельных компонентов для формирования более сложных потоков сообщений. В отличие от него брокер сообщений изначально касается крупных решений и направлен на борьбу с неизбежной сложностью управления такими системами.

Брокер сообщений не является цельным, неделимым компонентом. Напротив, он имеет сложную внутреннюю структуру, в которой используется целый ряд шаблонов маршрутизации, представленных выше. Решив использовать брокер сообщений в качестве архитектурного шаблона, для его реализации можно выбрать любую подходящую разновидность или разновидности маршрутизатора сообщений.

Преимущество единой точки поддержки, которую обеспечивает брокер сообщений, может превратиться в его недостаток. Если все сообщения системы проходят через один брокер сообщений, последний почти наверняка будет представлять собой ‘‘узкое место’’. Существует несколько приемов, позволяющих обойти эту проблему. Прежде всего отметим, что брокер сообщений как архитектурный шаблон подразумевает лишь разработку одного компонента, выполняющего маршрутизацию. В нем не содержится никаких указаний насчет того, сколько экземпляров этого компонента может быть развернуто в системе с целью эксплуатации. Если структура брокера сообщений не предусматривает сохранение состояния (т.е. если он состоит исключительно из компонентов, работающих без сохранения состояния), то можно безо всяких опасений развернуть несколько экземпляров брокера для улучшения пропускной способности системы. Свойства канала ‘‘точка-точка’’ гарантируют, что каждое входящее сообщение будет потреблено только одним экземпляром брокера сообщений. Кроме того, в реальной жизни итоговое решение почти всегда оказывается комбинацией шаблонов. Аналогичным образом во многих сценариях интеграции имеет смысл разработать несколько разных брокеров сообщений, каждый из которых будет специализироваться на небольшой порции решения. Это позволит избежать создания некоего ‘‘сверхброкера сообщений’’, структура которого окажется непомерно сложной для управления и поддержки. Обратной стороной описанного подхода, очевидно, является потеря единой точки поддержки и потенциальное появление новой разновидности интеграционного ‘‘спагетти’’, на сей раз образованного многочисленными брокерами сообщений. В качестве превосходного архитектурного стиля, в котором используется комбинация брокеров сообщений, можно предложить иерархию брокеров сообщений.



Поскольку назначение брокера сообщений состоит в том, чтобы максимально ослабить связь между отдельными приложениями, ему обычно приходится иметь дело с преобразованием сообщений из одного формата данных в другой. Абстрагирование брокером маршрутизации сообщений никак не поможет приложению-отправителю, если тому необходимо отослать сообщение в (предположительно неизвестном) формате приложения-получателя. Для решения этих проблем применяется ряд шаблонов преобразования сообщений, о которых будет рассказываться далее. В частности, довольно часто внутри брокера сообщений используется каноническая

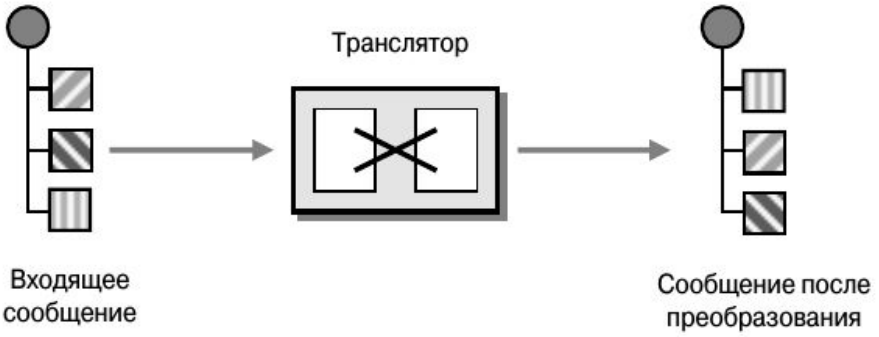
модель данных, позволяющая избежать знаменитой проблемы N^2 (с увеличением числа получателей системы число трансляторов, необходимое для преобразования сообщений между каждой парой получателей, растет в квадрате).

Преобразователи сообщений (Message Translators)

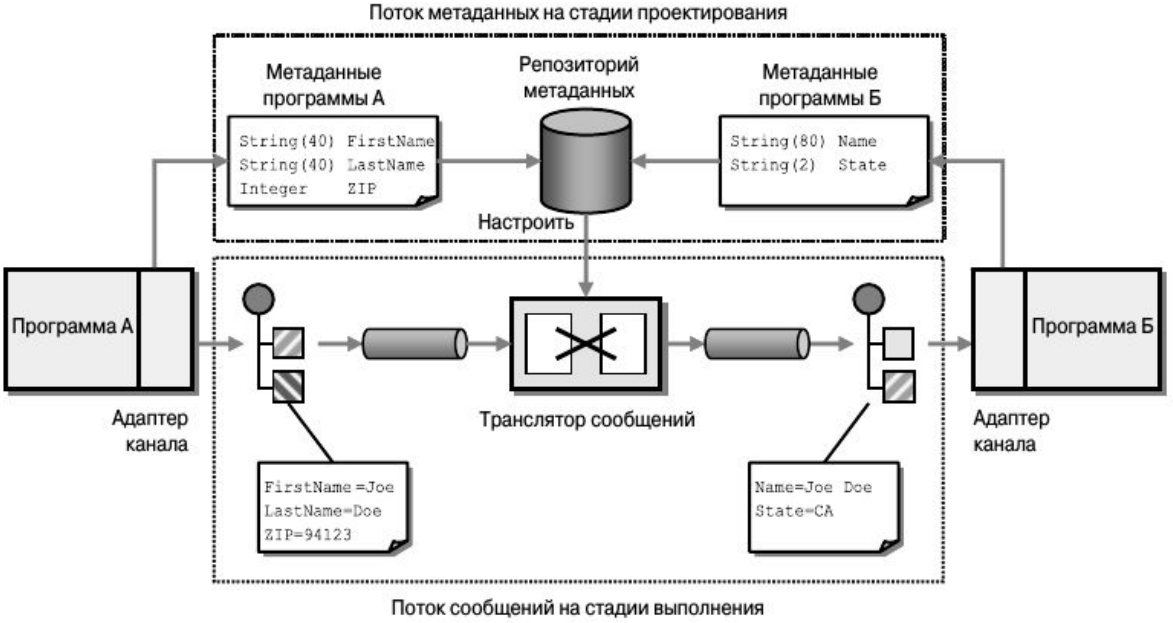
Как мы уже говорили ранее, большинство интеграционных решений объединяет разнородные приложения -- legacy-системы, коммерческие и созданные на заказ. Как правило, каждое из этих приложений использует собственную модель данных. К примеру, система бухгалтерского учета может оперировать такими данными о заказчике, как номер налогоплательщика, а система управления взаимоотношениями с клиентами (CRM) -- телефонным номером и адресом проживания.

Функция преобразования формата данных может быть встроена в конечную точку сообщения (Message Endpoint). В результате все приложения будут отправлять и принимать сообщения общего формата данных. Однако этот подход предполагает наличие доступа к исходному коду конечной точки, что возможно далеко не всегда. К тому же встраивание кода преобразования в конечную точку снижает возможность его повторного использования.

В таких случаях для преобразования формата данных специальный фильтр -- преобразователь сообщений, расположив его между другими фильтрами или приложениями. Транслятор сообщений -- эквивалент шаблона проектирования Адаптер, который преобразует интерфейс компонента в другой интерфейс, который может быть использован в отличном контексте.



Преобразование сообщений из одного формата в другой выполняется путем обработки метаданных (служебной информации, описывающей фактический формат данных сообщения). Метаданные играют настолько важную роль в интеграции корпоративных приложений, что большинство интеграционных решений можно рассматривать как взаимодействие двух параллельных систем. Одна из систем будет работать с фактическими данными сообщениями, а другая с метаданными. Многие шаблоны, используемые для формирования потока данных сообщений, могут применяться и для управления потоком метаданных. Например, адаптер канала способен не только перемещать сообщения между приложением и системой обмена данными, но и извлекать метаданные из внешних приложений, загружая их в центральный репозиторий метаданных. Используя указанный репозиторий, разработчики интеграционных решений могут определять преобразование метаданных приложения в каноническую модель данных.



Рассмотрим несколько типов преобразователей.

Упаковщик

Данный преобразователь позволяет упаковать/распаковать данные приложения в формат, совместимый с требованиями инфраструктуры обмена сообщениями. К примеру, в этот формат может входить специального вида заголовок с набором служебных полей, да и вообще конечному приложению про эти детали знать совсем необязательно.

В сложных инфраструктурах несколько упаковщиков могут применяться друг за другом. В результате оказывается, что полезная информация внутри сообщения содержит еще одно сообщение, которое, в свою очередь, включает в себя заголовок и полезную информацию.

Фильтр содержимого

В зависимости от ситуации преобразователи могут захотеть как-то изменить содержимое сообщения. Например, преобразовать формат дат внутри, добавить индекс по имеющемуся адресу или, наоборот, удалить какую-нибудь избыточные или закрытые данные. Фильтры также могут упрощать структуру сообщений.

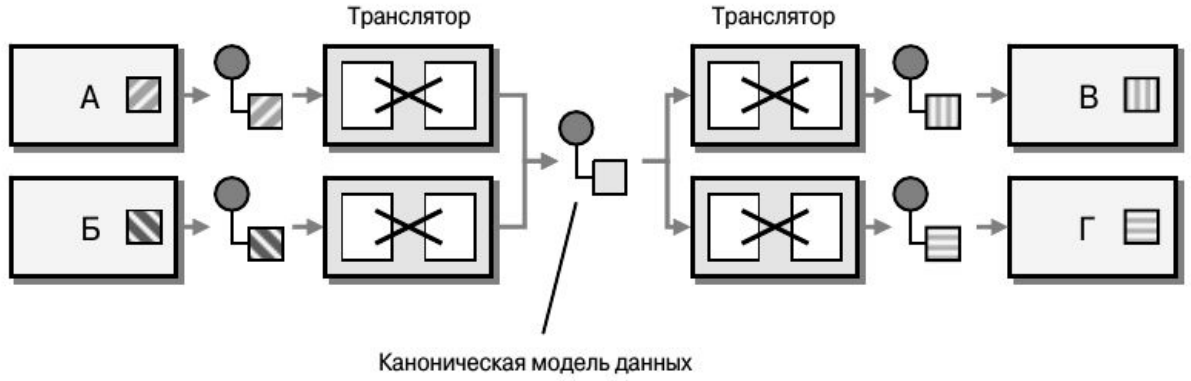
Каноническая модель данных

Преобразователи сообщений позволяют устранить проблемы, связанные с различием форматов сообщений, без необходимости вносить изменения в логику приложений или уведомлять их о формате данных друг друга. Но если в обмене данными принимает участие сразу несколько приложений, на каждую пару участников понадобится по отдельному транслятору сообщений.

Единая внутренняя модель данных обеспечивает появление дополнительного уровня косвенности между отдельными форматами данных, используемыми в приложениях. Если в интеграционном решении появится новое приложение, разработчику придется добавить всего лишь одно-два преобразования между

приложением и канонической моделью данных вне зависимости от того, сколько приложений уже участвует в обмене данными.

Использование канонической модели данных может показаться лишним, если интеграционное решение охватывает небольшое число приложений. Данный метод, однако, очень быстро оправдывает себя с возрастанием их количества.

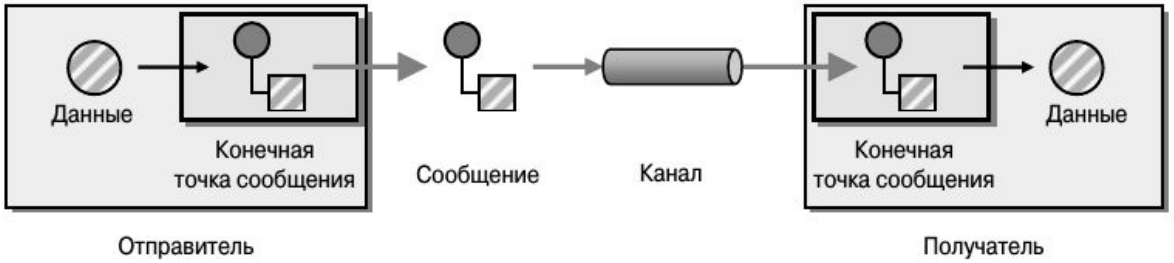


При использовании канонической модели данных время обработки потока сообщений увеличивается. Каждому сообщению вместо одного преобразования теперь приходится претерпевать целых два: одно преобразование из формата исходного приложения в общий формат и еще одно из общего формата в формат приложения-получателя. По этой причине использование канонической модели данных часто называют двойной трансляцией (double translation), а преобразование из формата одного приложения непосредственно в формат другого прямой трансляцией (direct translation). Таким образом, для систем, требующих очень быстрого отклика, единственным допустимым решением остается прямая трансляция. В реальной жизни при выборе способа преобразования сообщений часто приходится жертвовать либо производительностью, либо гибкостью.

Конечные точки сообщений (Message Endpoints)

Приложение и система обмена сообщениями представляют собой две отдельные программные сущности. Приложение обеспечивает функциональность для пользователей, в то время как система обмена сообщениями управляет каналами, применяющимися для передачи сообщений. Даже будучи встроенной в приложение, система обмена сообщениями предоставляет обособленные, специализированные функции подобно СУБД или Web-серверу. В связи с этим возникает задача подключения приложения к системе обмена сообщениями.

Система обмена сообщениями представляет собой определенный тип сервера, принимающего запросы и отвечающего на них. Клиентом сервера сообщений является приложение, взаимодействующее с другими приложениями с помощью обмена сообщениями. Подобно серверу баз данных сервер сообщений предоставляет клиентский API, с помощью которого приложение может взаимодействовать с сервером. Поскольку клиентский API сервера сообщений отражает специфику конкретной системы обмена сообщениями, приложение должно включать в себя код подключения к системе обмена сообщениями.



Код конечной точки сообщений создается с учетом особенностей конкретного приложения и клиентского API системы обмена сообщениями. Оставшаяся часть приложения не обладает сведениями о формате сообщений, каналах и других подробностях взаимодействия посредством обмена сообщениями. Конечная точка сообщения инкапсулирует систему обмена сообщениями от приложения. Таким образом, внесение изменений в приложение или в клиентский API системы обмена сообщениями приведет к необходимости правки кода только конечной точки сообщений.

Конечная точка сообщения является частным случаем одновременно адаптера канала и преобразователя сообщений, интегрированного в приложение и созданного для его подключения к конкретной системе обмена сообщениями.

Среди функциональности, которую могут реализовывать конечные точки сообщений, можно выделить следующую:

- инкапсуляция кода, относящегося к отправке и получению сообщений, и отделение его от остального кода приложения;
- фильтрация сообщений;
- поддержка транзакционности операций;
- реализация событийной или синхронной схем работы с сообщениями;
- реализации конкурентной обработки входящих сообщений;
- диспетчеризация сообщений между потребителями;

Особенности коммуникаций через обмен сообщениями

Итак, под конец сформулируем основные достоинства и недостатки обмена сообщениями перед другими технологиями интеграции приложений.

- **Удаленное взаимодействие**. Обмен сообщениями позволяет наладить взаимодействие между отдельными приложениями. Два объекта, относящихся к одному и тому же процессу, могут совместно использовать данные, размещенные в оперативной памяти. Передача информации с одного компьютера на другой требует выполнения сериализации данных. Приложение может делегировать всю ответственность за передачу данных системе обмена сообщениями, тем самым избавляясь от части сложной функциональности.
- **Платформенная/языковая интеграция**. Зачастую удаленные системы создаются с использованием различных платформ, технологий и языков программирования. Интеграция разнородных систем требует использования связующего ПО, в качестве которого может выступить система обмена сообщениями. Идея использования системы обмена сообщениями в качестве

универсального связующего звена между приложениями была положена в основу шаблона Шина сообщений (Message Bus).

- **Асинхронное взаимодействие**. Обмен сообщениями позволяет наладить взаимодействие между приложениями по принципу “отправил и забыл”. В соответствии с этим принципом отправитель не обязан ожидать подтверждение о получении и обработке сообщения от принимающей стороны. Более того, он также не обязан ожидать подтверждение о доставке сообщения от системы обмена сообщениями. Единственное, о чем следует позаботиться отправителю, -- это дождаться подтверждения об отправке сообщения, т.е. о его помещении в канал. Как только сообщение будет передано системе обмена сообщениями, отправитель может приступить к выполнению имеющихся у него задач.
- **Рассогласование во времени**. При синхронном взаимодействии отправитель должен дождаться завершения обработки вызова получателем прежде, чем сделать новый вызов. Таким образом, скорость размещения вызовов отправителем ограничена скоростью их обработки получателем. Асинхронное взаимодействие позволяет размещать и обрабатывать вызовы с разной скоростью, что существенно повышает эффективность взаимодействия между приложениями.
- **Регулирование нагрузки**. Слишком большое число удаленных вызовов процедур за короткий промежуток времени может привести к перегрузке получателя, снижению его производительности и даже выходу из строя. Система обмена сообщениями формирует очередь запросов, позволяя получателю контролировать скорость их обработки. Поскольку взаимодействие осуществляется в асинхронном режиме, регулирование нагрузки на стороне получателя не оказывает негативного влияния на отправителя.
- **Надежное взаимодействие**. В отличие от удаленного вызова процедуры, обмен сообщениями позволяет наладить надежное взаимодействие между приложениями за счет подхода, получившего название “передача с промежуточным хранением” (store-and-forward). Как упоминалось выше, сообщение представляет собой единицу передачи информации, инкапсулирующую полезные данные. Когда отправитель помещает сообщение в канал, система обмена сообщениями сохраняет его на компьютере отправителя. Затем сообщение доставляется получателю и сохраняется на его компьютере. Предположим, что сохранение сообщения на компьютерах отправителя и получателя является надежной операцией. (Чтобы сделать ее еще надежнее, сообщение можно сохранять не в памяти, а на диске компьютера.) Единственным слабым звеном передачи с промежуточным хранением является доставка сообщения на компьютер получателя. Чтобы нивелировать негативное влияние возможных сбоев при передаче сообщения, система обмена сообщениями пересылает его до тех пор, пока оно не будет доставлено по назначению. Автоматическая пересылка сообщения позволяет исключить риск потери информации при возникновении сбоя в сети или на компьютере получателя, что, в свою очередь, делает возможным применение принципа взаимодействия между приложениями ‘‘отправил и забыл’’.

- **Работа без подключения к сети**. Некоторые приложения ориентированы на работу без подключения к сети. Как правило, подобные приложения предназначены для выполнения на ноутбуках или мобильных устройствах и периодически (при наличии сетевого подключения) синхронизируют данные с сервером. Обмен сообщениями -- идеальное решение для синхронизации, позволяющее накапливать данные в очереди до тех пор, пока приложение не получит доступ к сети.
- **Посредничество**. Система обмена сообщениями выступает в роли посредника (Mediator) между взаимодействующими приложениями. Приложение может использовать систему обмена сообщениями в качестве каталога доступных для интеграции приложений и служб. Если приложение потеряет связь с другими приложениями, ему понадобится восстановить соединение только с системой обмена сообщениями, а не с каждым отдельным приложением. Функция посредника может потребовать от системы обмена сообщениями высокой доступности, балансировки нагрузки, устойчивости к отказам сетевых соединений, а также поддержки качества обслуживания.
- **Управление потоками**. Асинхронное взаимодействие позволяет приложению не ожидать результата выполнения задачи другим приложением. Вместо этого приложение может воспользоваться обратным вызовом, уведомляющим его о поступлении ответа. Большое число заблокированных потоков, а также потоки, заблокированные в течение длительного времени, могут оказать негативное воздействие на работу приложения. Кроме того, такие потоки трудно восстановить в случае сбоя приложения и его последующего перезапуска. Использование обратного вызова позволяет минимизировать количество заблокированных потоков, обеспечить стабильную работу приложения и определить потоки, которые должны быть восстановлены при его перезапуске.

Итак, некоторые из описанных выше преимуществ обмена сообщениями перед другими способами организации взаимодействия приложений носят сугубо технический характер, в то время как остальные представляют собой стратегические решения, принимающиеся на этапе проектирования приложения. Безусловно, каждое из вышеперечисленных преимуществ обмена сообщениями будет иметь различный вес в контексте конкретных требований, предъявляемых к приложению.

Но несмотря на то, что технология асинхронного обмена сообщениями позволяет преодолеть множество трудностей, связанных с интеграцией разнородных приложений, она не лишена недостатков. Некоторые из них являются неотъемлемой частью асинхронной модели взаимодействия, в то время как остальные зависят от конкретной реализации системы обмена сообщениями.

- **Сложная модель программирования**. Асинхронный обмен сообщениями требует от разработчиков использования модели событийно управляемого программирования. В этом случае логика приложения разбивается на множество обработчиков событий, реагирующих на входящие сообщения. Подобную систему гораздо труднее программировать и отлаживать, чем систему, основанную на вызове методов. Так, эквивалентом простого вызова метода в модели событийно управляемого программирования является совокупность, состоящая из сообщения с запросом, канала запроса, сообщения

с ответом, канала ответа и какой-нибудь очереди сообщений недопустимого формата.

- **Порядок доставки сообщений**. Система обмена сообщениями гарантирует доставку сообщения от отправителя к получателю, не оговаривая требующееся для этого время. В результате может быть нарушен порядок, в котором были отправлены сообщения. Если последовательность доставки сообщений имеет значение, ее нужно восстановить (например, с помощью специального компонента-преобразователя).
- **Необходимость реализации синхронной модели**. Не все приложения могут взаимодействовать по принципу ‘‘отправил и забыл’’. К примеру, пользовательский запрос о наличии авиабилетов должен быть обработан немедленно, а не в течение неопределенного промежутка времени. Следовательно, в некоторых системах обмена сообщениями должен быть предусмотрен баланс между синхронной и асинхронной моделями взаимодействия.
- **Производительность**. Системы обмена сообщениями вносят дополнительные издержки в процесс взаимодействия между приложениями. На создание, отправку, получение и обработку сообщения уходят время и ресурсы. К тому же передача большого объема данных может повлечь за собой создание несметного числа сообщений. Так, зачастую интеграция двух существующих систем начинается с репликации всех необходимых данных. Репликация большого объема информации с помощью средств ETL (Extract, Transform and Load) гораздо эффективнее репликации с помощью обмена сообщениями. Таким образом, обмен сообщениями рекомендуется применять для синхронизации данных между приложениями, а не для их первичной репликации.
- **Зависимость от компании-разработчика**. Коммерческие системы обмена сообщениями могут основываться на использовании закрытых протоколов. Открытые системы в этом смысле безопаснее в использовании, но переход с одного открытого MOM на другой тоже может потребовать кучи ресурсов. Да и различные системы обмена сообщениями часто оказываются неспособными к взаимодействию друг с другом. Это может привести к возникновению новой задачи интеграции: интеграции нескольких различных интеграционных решений!

Литература

1. Г. Хоп, Б. Вульф. Шаблоны интеграции корпоративных приложений
2. М.Фаулер. Архитектура корпоративных программных приложений