Повышение доступности клиентских приложений при разрывах сетевых соединений

В. О. Демиш[1,2], Б. Н. Пищик[1,2,*]

[1]Конструкторско-технологический институт вычислительной техники СО РАН, Новосибирск, Россия

[2]Новосибирский государственный университет, Россия

[*]Контактный e-mail: boris.pishchik@kti.nsc.ru

Рассмотрена проблема устойчивости клиентских приложений, работающих с веб-сервисами посредством программного интерфейса CRUD, к разрывам сетевых соединений. Предложен алгоритм обеспечения отложенной согласованности данных, реализуемый целиком на клиентском приложении без изменения программного интерфейса веб-сервиса. Алгоритм позволяет обеспечить работу части функций клиентского приложения при недоступности веб-сервиса. Рассмотрены проблемы хранения и синхронизации данных на стороне клиента.

Ключевые слова: клиент-серверные приложения, разрыв сетевого соединения, повышение доступности.

Введение

Количество различных сервисов в сети Интернет, предоставляющих доступ к своим функциям посредством интерфейса программирования приложений (API), непрерывно растет. Такие сервисы часто включают работу с данными, и в этих случаях API предполагает наличие ставших типовыми методов чтения, изменения, добавления и удаления данных. Это, например, сервис организации информации Evernote (evernote.com), CRM-системы SaleForces (www.salesforce.com/crm/), Zoho CRM (www.zoho.com), сервисы Google (www.google.ru/intl/ru/about/products/), Yandex (www.yandex.ru/all), сервисы социальных сетей и др. Если API сервиса не предполагает распределенной работы с приложениями, которые его используют, то чаще всего такие приложения являются своего рода “онлайн-клиентами” для работы с сервисом. В настоящей работе исследуется возможность обеспечить клиентское приложение частью функций, которые будут оставаться работоспособными при недоступности сервиса.

1. Шаблоны проектирования API

Разработка API для информационной системы — комплексная задача, сочетающая в себе решение таких вопросов, как выбор оптимальных форматов данных, обработка ошибок, выбор подходящих протоколов, проектирование методов интерфейса и т. д. Существуют различные шаблоны (и антишаблоны) проектирования API [1, 2], в различной мере рассматривающие эти вопросы.

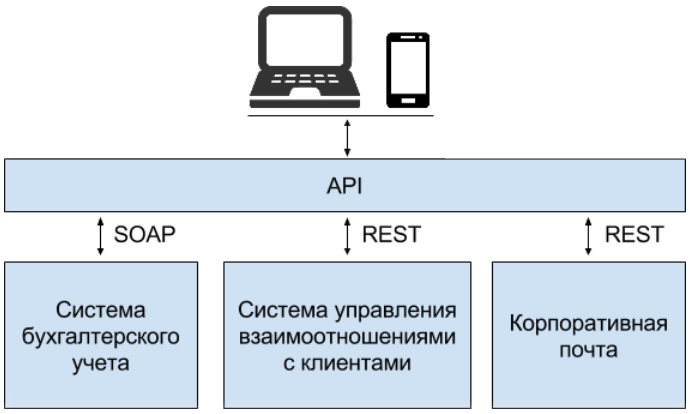

Рис. 1. Шаблон проектирования Facade

Для многокомпонентной информационной системы чаще всего рекомендуется использование шаблона Facade (рис. 1), название которого происходит от архитектурного термина “фасад” — лицевая часть здания. Суть этого шаблона проектирования — создание интерфейса, который является не просто суммированием доступных интерфейсов каждого компонента системы, а представляет собой некоторое обобщение над всеми компонентами. Например, для информационной системы с компонентами “Процессор”, “Память”, “Жесткий диск” спроектированный по шаблону Facade интерфейс даст возможность работать с обобщением “Компьютер”, а не только с каждым компонентом в отдельности.

Не рассматривая все тонкости разработки API, в дальнейшем будем предполагать, что посредством API мы работаем с монолитной (с некоторой точки зрения) информационной системой. Фактически она может быть представлена различными компонентами. При этом также не обязательно, чтобы весь представленный в компонентах функционал был доступен посредством API.

Используя опыт проектирования API [3, 4], будем считать, что в составе программного интерфейса CRUD доступны операции *Create*, *Read*, *Update*, *Delete*, реализованные в виде следующих методов:

- *Get* — получение списка объектов и их атрибутов (возможно не всех), характеризуемых набором свойств (атрибутов), по заданному условию отбора;
- *Insert* — создание нового объекта с заданными значениями свойств;
- *Update* — обновление значений свойств объекта по его идентификатору;
- *Delete* — удаление объекта по его идентификатору.

Детали реализации этих методов не являются принципиальными в настоящей работе, поэтому в общем случае можно считать, что значения атрибутов передаются в виде ассоциативных массивов из пар “атрибут => значение”. В операции Get набор списка получаемых объектов задается в виде множества, в котором перечислены нужные атрибуты, а условием отбора является булевозначная формула с атрибутами в качестве пропозициональных переменных.

2. Структура информационной системы клиента

Рассматривая API как инструмент работы с различными типами объектов, можно считать, что имеется набор объектов различных типов. Объекты определяются совокуп-

ностью атрибутов и их значений (атомарных). При этом предполагается наличие атрибута, однозначно идентифицирующего объект среди других объектов того же типа (ключ).

В дальнейшем будем считать, что структура данных на сервере остается неизменной, операции по изменению структуры в API отсутствуют (соответствует отсутствию операции ALTER TABLE для реляционных баз данных).

В зависимости от того, используется ли в клиентском приложении собственное информационное хранилище, работа с API может быть выстроена двумя способами:

1. "Тонкий клиент". Вызывая методы API, клиентское приложение сразу отображает результат операции. Такое клиентское приложение работает с сервером (той его частью, которая доступна через API) в режиме онлайн. Как только сервер становится недоступен, приложение вынуждено прекратить работу.
2. "Толстый клиент". Работая с сервером через API, клиентское приложение может сохранять часть информации об отношениях в собственном информационном пространстве (далее — в локальной БД). Это позволит предоставить часть функций после того, как сервер станет недоступным.

В первом случае клиентское приложение изначально не является устойчивым к разрывам соединений (отсутствует устойчивость к разделению), поскольку при недоступности сервера оно не может функционировать. Во втором варианте возможности клиента шире. В настоящей работе предложен алгоритм обеспечения устойчивости к разрывам сетевых соединений в ущерб согласованности данных. Иными словами, реализуется вариант "доступность и устойчивость к разделению" вместо "доступность и согласованность" в соответствии с терминологией теоремы CAP [5]. В этом варианте согласованность данных замещается отложенной согласованностью, которая обеспечивается через некоторый период времени, допуская несогласованность данных на этот период. Реализация такого варианта предполагает наличие локальной базы данных в клиентском приложении.

Для обеспечения отложенной согласованности значения одних атрибутов имеет смысл актуализировать как можно быстрее, другие же могут быть не так важны. Например, для пользовательского приложения, в функции которого входит публикация записей пользователей, может быть допустима некоторая задержка в обновлении информации о количестве просмотров записей другими пользователями. Но обновление заголовков в ленте новостей может быть критичным — свежие новости должны оперативно отображаться в приложении.

Чтобы наделить клиентское приложение разнообразием возможностей работы с различными атрибутами, значения которых считываются с сервера, их можно разделить на четыре группы с точки зрения важности атрибутов, оцениваемой на уровне бизнес-логики клиентского приложения. Это можно сделать дополнительным параметром — показателем актуальности, принимающим значения от 0 до 3:

0 — атрибут локальный, и его значение не хранится на сервере;

1 — преимущественно использовать локальное значение;

2 — преимущественно использовать серверное значение;

3 — всегда использовать серверное значение.

В дальнейшем будем обозначать этот параметр RI (от *Relevance Indicator*), а также считать, что для каждого атрибута его значение зафиксировано. С учетом его значений можно осуществлять выполнение рассмотренных ранее методов API в несколько шагов таким образом, чтобы каждое обращение к серверу обновляло значения атрибутов

в локальной БД. В случае недоступности сервера необходимо использовать значения из локальной БД и фиксировать невыполненный запрос к серверу для его повторного вызова в дальнейшем.

Таким образом, добавление локальной БД изменяет (усложняет) протокол взаимодействия клиента и сервера. Изначально каждая операция вызывает метод API сервиса, который возвращает запрашиваемые данные и информацию о результате выполнения операции либо сообщение об ошибке (в данном случае не рассматриваются ошибки, вызванные потерей соединения с сервером). После добавления локальной БД в клиентское приложение выполнение каждой операции API включает дополнительные действия на клиенте. Для удобства введем новые обозначения — чтобы отличить исходные операции от операций, использующих локальную БД. Последние будем помечать знаком “*”, например — Get^*, а $Insert^*$, $Update^*$ и $Delete^*$ — аналогично.

Новый порядок работы для Get^* с использованием локальной БД показан на рис. 2. Блоки выполнения, выделенные серым цветом, при недоступности сервера помещают-

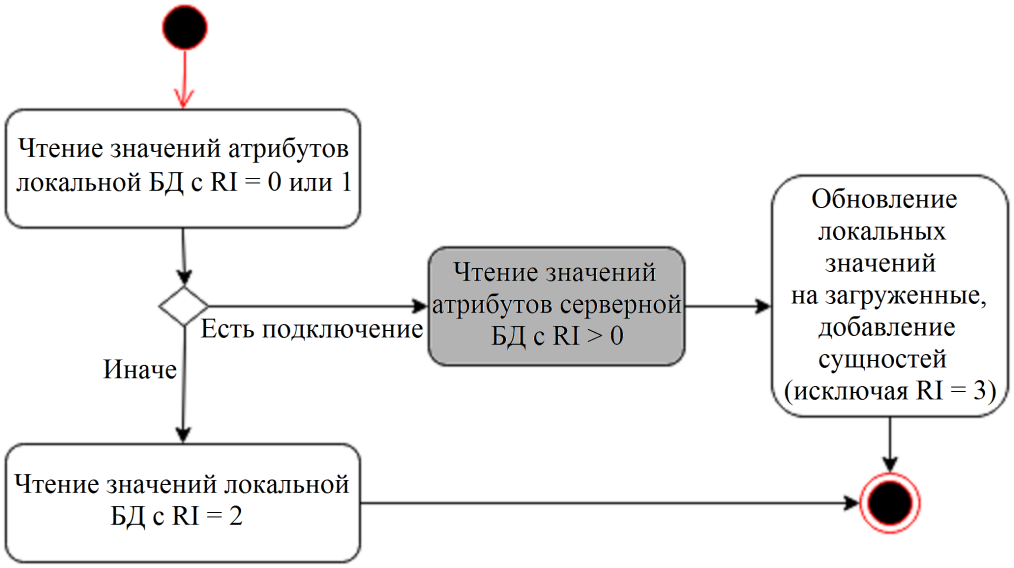

Рис. 2. Операция Get^*

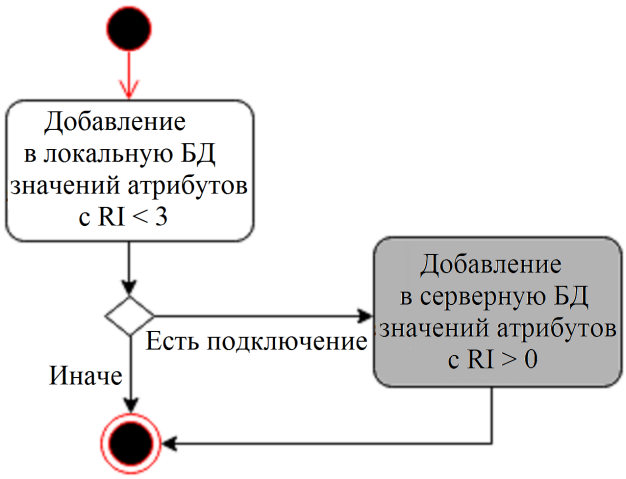

Рис. 3. Операции $Insert^*$, $Update^*$, $Delete^*$

ся в очередь отложенных вызовов (см. далее). Для операции Get^* после выполнения отложенного (из очереди) вызова должно произойти обновление данных локальной БД (следующий блок на схеме), а для операций $Insert^*$, $Update^*$ и $Delete^*$ — выполнение соответствующих операций манипулирования данными на сервере (рис. 3).

Таким образом, при последующем подключении к серверу выполнение отложенных вызовов производит синхронизацию локальной и серверной БД при условии отсутствия конфликтов (изменении одних и тех же данных разными клиентами).

Поскольку объекты могут создаваться как в клиентском приложении, так и на сервере, предполагается, что все объекты клиента содержат как серверное, так и собственное значение первичного ключа. В качестве основного используется серверное значение, а до момента отправки объекта на сервер (и получения соответствующего ключа) используется временное локальное значение.

3. Очереди отложенных вызовов

При временной недоступности сервера все вызовы серверных методов API на клиентском приложении помещаются в очередь. Для каждой сущности имеется своя очередь отложенных вызовов. В дальнейшем, при подключении к серверу, все команды из очереди выполняются в порядке их помещения в очередь.

Очевидной проблемой такого подхода является неизбежное переполнение очереди в случае длительной недоступности сервера. Решить эту проблему можно как минимум двумя способами:

- ограничить объем очереди (достижение которого остановит работу приложения);
- уменьшить очередь до другой, эквивалентной ей, не останавливая работу приложения.

Если первый способ не вызывает вопросов, то второй требует уточнений. Определим понятие эквивалентности очередей. Две очереди команд являются *эквивалентными*, если их выполнение приведет к идентичным преобразованиям значений атрибутов сущностей на серверной и клиентской БД. Поскольку в очереди могут быть операции лишь четырех видов (которые доступны посредством API), способы сжатия очереди целесообразно рассмотреть для каждой из них. Для этого рассмотрим добавление в очередь нового вызова операции API и преобразование очереди с целью уменьшения ее объема. Будем считать, что компьютер клиента и сервер не содержат конфликтующих данных (разрешение конфликтов — это отдельная задача). Также, без потери общности, будем считать, что в системе представлен один тип объектов с фиксированным набором атрибутов.

Замечание 1. В зависимости от специфики приложений, конкретных характеристик клиентских и серверных устройств, а также возможностей информационного канала между клиентом и сервером уменьшение объема очереди в клиентском приложении может не иметь существенного значения. Более важным может быть уменьшение объемов трафика между клиентскими устройствами и сервером, а также оптимизация выполнения клиентских запросов на сервере. Однако в настоящей работе предполагается существенная ограниченность ресурсов на клиентском устройстве, что указывает на необходимость преобразований очереди отложенных операций.

Алгоритм добавления команды в очередь состоит из шагов, которые могут включать следующие этапы: анализ операций в очереди, преобразование существующих операций либо добавляемой операции, вставка операции в очередь, удаление операций из очереди.

Объектом оптимизации является количество отложенных операций в очереди, которое не должно увеличиваться в результате преобразований. Если очередь пустая, то вне зависимости от типа операции ее отложенный вызов просто добавляется в очередь без каких-либо преобразований.

Рассмотрим этапы добавления отложенных вызовов операций в непустую очередь.

1. Insert. Добавление $Insert$ включает лишь один этап — вставку в очередь.

2. Delete. Добавление $Delete$ убирает из очереди все операции, которые относились к изменению удаляемой сущности, — все $Update$, а также $Insert$. При этом вставка операции $Delete$ в очередь не осуществляется.

3. Update. Предположим, происходит добавление команды $Update(Id, A, V)$, где Id — идентификатор изменяемого объекта (первичный ключ), A — список изменяемых атрибутов, а V — новые значения.

Для проведения преобразования очереди необходимо проанализировать другие операции в очереди. Поскольку $Delete$ не может присутствовать в очереди (изменение уже удаленного объекта невозможно), нужно рассмотреть взаимодействие с тремя другими видами операций.

- $Insert(V)$. Если изменению объекта предшествует его создание, то значит, такой объект еще не был выгружен на сервер. Следовательно, на сервер может быть отправлена одна операция $Insert$ с финальными значениями атрибутов объекта. Для этого необходимо в операции $Insert$ изменить значения атрибутов добавляемого объекта на более актуальные, которые были переданы в операции $Update$. Добавления самой операции $Update$ в данном случае не происходит.
- *Update.* Если в очереди обнаружена операция $Update$, значит, объект уже отправлен на сервер (в противном случае в очереди был бы найден $Insert$ и было бы выполнено преобразование, описанное выше). Обозначим обнаруженную операцию $Update(Id, A', V')$. По аналогии с $Insert$ идея преобразования заключается в объединении двух операций в одну с указанием актуальных значений атрибутов, которые должны быть у объекта после применения обеих операций $Update$. Иными словами, $Update(Id, A, V)$ и $Update(Id, A', V')$ преобразуются в одну операцию $Update(Id, A'', V'')$, где $A'' = A \cup A'$ и V'' сформированы из значений V и V' с предпочтением значений V при изменении одних и тех же атрибутов. Далее осуществляется вставка полученной таким образом $Update(Id, A'', V'')$ в конец очереди, а обнаруженная ранее операция $Update(Id, A', V')$ удаляется. Если после $Update(Id, A', V')$ в очереди были операции Get, то описанное преобразование может привести к неэквивалентной очереди. $Update$ и Get не являются перестановочными с точки зрения сохранения эквивалентности. Чтобы сохранить эквивалентность, необходимо соблюдать порядок операций — все операции изменения данных должны предшествовать операциям чтения (см. взаимодействие с Get далее). Для этого все операции Get, найденные между $Update(Id, A', V')$ и концом очереди, должны быть сдвинуты в конец очереди.
- *Get.* Каждое серверное чтение должно в итоге обновлять (или создавать) прочитанные значения с сервера в локальной базе. Из-за того, что часть данных будет изменена добавляемой операцией $Update$, полученные с сервера данные будут содержать избыточную информацию при условии, что $Update$ предшествовал операции Get. Если же порядок обратный, т. е. на сервер сначала отправляется вызов операции Get, а затем $Update$, то полученная в клиентское приложение с помощью Get информация будет некорректной, поскольку на клиентской стороне уже

Типы преобразования операции Get

Тип 1	Тип 2
$Get(A_1 \cap A_2, R_1 \vee R_2)$ $Get(A_1 \backslash A_2, R_1)$ $Get(A_2 \backslash A_1, R_2)$	$Get(A_1 \cup A_2, R_1 \vee R_2)$

выполнен $Update$, который на момент выполнения Get еще не был отправлен на сервер. Во избежание подобных проблем необходимо, чтобы все операции чтения осуществлялись после операций изменения данных.

Таким образом, наличие операций Get в очереди при добавлении $Update$ требует соблюдения указанного порядка — $Update$ должен быть добавлен не в конец очереди, а предшествовать всем операциям чтения.

4. Get. Исходя из уже рассмотренного взаимодействия различных операций можно считать, что все операции Get расположены после операций $Insert$, $Update$, $Delete$. Можно также ограничиться рассмотрением взаимодействия с другими операциями Get, поскольку остальные случаи будут аналогичны рассмотренным.

Идея уменьшения количества вызовов сервера заключается в том, чтобы не считывать одни и те же значения несколько раз. Пусть исходная очередь содержит операции $Get_1(A_1, R_1)$ и $Get_2(A_2, R_2)$. Рассмотрим два типа преобразований очереди, в ходе которых эти операции либо преобразуются в три операции (первый тип преобразований), либо объединяются в одну (второй тип преобразования) (см. таблицу).

Утверждение 1. В результате преобразований первого типа получившаяся очередь эквивалентна исходной.

Утверждение 2. Второй тип преобразований не всегда приводит к очереди, эквивалентной исходной. Количество операций в ней будет уменьшено, но с сервера, возможно, будет считано больше значений атрибутов, чем требовалось до преобразования.

Если $A_1 = A_2$, применение преобразования первого типа приведет к появлению одной команды, а не трех, поскольку $A_1 \backslash A_2$ и $A_2 \backslash A_1$ в этом случае будут пустыми. На практике это означает просмотр списка объектов одного типа — будут отбираться одни и те же атрибуты, но с различными условиями.

Повсеместное применение преобразования второго типа в итоге объединит множество команд Get в одну с дизъюнкцией всех условий отбора R. Это существенно уменьшает количество операций чтения (до одной итоговой).

Без уточнения деталей реализации условий R достаточно сложно сказать что-либо о целесообразности применения того или иного типа преобразований. Может получиться, что выполнение дизъюнкции будет слишком трудоемким и тем самым существенно замедлит работу клиентских приложений. Для различных платформ и средств разработки задание условий может быть реализовано множеством способов. Например, платформа 1С-Битрикс “Управление сайтом” предлагает использование ассоциативных массивов PHP с их объединением в логические блоки И и ИЛИ [6].

4. Синхронизация данных на клиенте и сервере

В исходной постановке задачи предполагается, что серверное приложение “не знает” о том, что на стороне клиентского приложения хранятся полученные ранее данные. С точки зрения сервера клиент всегда работает онлайн. Таким образом, результат из-

менения или удаления объекта на сервере может быть незаметен для клиентского приложения.

Для достижения отложенной согласованности данных на сервере и в клиентском приложении необходима их синхронизация. Взаимодействие с сервером осуществляется посредством доступных операций API, выполнение которых должно включать дополнительные действия для обеспечения синхронизации данных. Обеспечение полной синхронизации данных (т. е. идентичности значений в серверной и клиентской базах данных) — это сложная комплексная задача. В настоящей работе предполагается, что локальная база данных может содержать не всю информацию, хранящуюся на сервере, а только ее часть, определяемую бизнес-логикой пользовательского приложения. Но эта часть должна быть в конечном счете синхронизирована с сервером, чтобы пользователи клиентского приложения работали с актуальной информацией.

Поведение пользователя, определяемое вызовами серверных методов API из клиентского приложения, может быть использовано для дальнейшего упрощения процедуры синхронизации данных, которая обеспечит актуальность информации в локальной базе данных. Клиентское приложение может удалять “сомнительные” данные из локальной базы, т. е. объекты, которые уже изменены или удалены на сервере, но их состояние все еще не актуализировано в локальной базе. Такое ослабленное требование к синхронизации данных позволяет упростить ее реализацию, не нарушая актуальности данных для пользователя (рис. 4).

Для того чтобы локальная база данных клиентского приложения была синхронизирована с сервером, необходимо выполнение дополнительных действий, которые будут разниться в зависимости от типа вызываемой операции:

- *Insert*. Вызов *Insert* обусловлен необходимостью проинформировать сервер о создании в клиентском приложении новых объектов. До обработки сервером этого вызова объект на стороне сервера отсутствует, поэтому заведомо не может быть изменен или удален.
- *Update*. Если объект был изменен на сервере, то его изменение на стороне клиента может спровоцировать конфликт изменения данных (при обращении к тем же атрибутам, значения которых изменились на сервере). Если в клиентском приложении изменился другой набор атрибутов, то конфликта не будет, однако с сервера

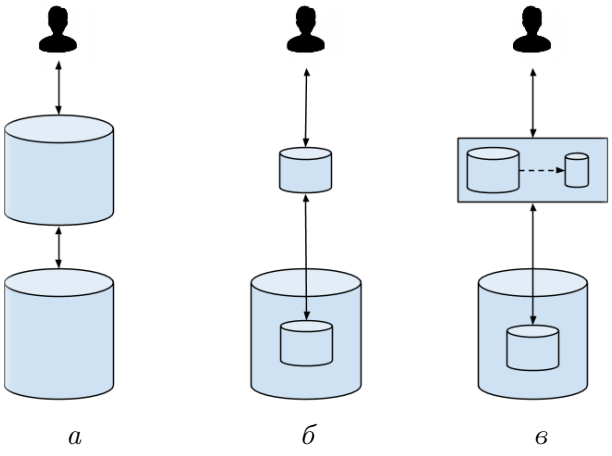

Рис. 4. Упрощение синхронизации данных от полной (*а*) к частичной (*б*) и регулируемой клиентским приложением (*в*)

в клиентское приложение должны быть загружены изменившиеся на сервере атрибуты, т. е. выполнен дополнительный *Get* для чтения атрибутов обновляемого объекта. В случае, если изменяемый в клиентском приложении объект был удален на сервере, то снова имеет место конфликт данных. Работа с подобными конфликтами — это задача, требующая отдельного рассмотрения, основные подходы к ее решению приведены в разд. 5.

- *Get*. Поскольку объекты на стороне сервера изменяются, выполнение одной и той же операции *Get* в разные периоды времени может давать различные результаты. В свою очередь, каждое выполнение *Get* предполагает обновление информации по объектам в локальной базе данных клиентского приложения. Таким образом, для того, чтобы определить, были ли удалены какие-то из загруженных ранее объектов, потребуется выполнить аналогичный *Get* на локальной базе данных клиентского приложения. Объекты, которые попали в выборку на локальной базе, но при этом не представлены в результате выполнения *Get* на сервере, будут либо изменены, либо удалены. В этом случае они также могут быть удалены из локальной базы данных.
- *Delete*. Выполнение команды *Delete* для объекта, который уже удален на сервере, вызовет ошибку. Однако это не помешает корректному выполнению синхронизации, поскольку в конечном счете объект оказывается удален как на сервере, так и на клиенте. В противном случае выполнение команды приводит к синхронизации состояния БД на сервере и в локальной БД клиента.

Замечание 2. Для обеспечения синхронизации данных не важно, был ли вызов операции API в режиме онлайн либо это был вызов отложенной операции из очереди в клиентском приложении. В обоих случаях результат синхронизации одинаков.

Предложенный вариант обеспечения упрощенной синхронизации данных аналогичен алгоритмам кэширования [7], в которых применяются различные стратегии выбора вытесняемых элементов. Вытеснение в данном случае соответствует удалению информации об объекте из локальной базы данных. Наиболее подходящая стратегия кэширования может быть применена на основе учета специфики приложения и ограничений на локальную базу данных.

5. Разрешение конфликтов

Поскольку изначально подразумевается, что клиент часто не может подключиться к серверу и разработчику приложения недоступны программные модули сервера, многие подходы, принятые для распределенных вычислений, неприменимы (например, использование блокировок и основанных на них алгоритмов) [8]. Поэтому приходится иметь дело с фактическими конфликтами обновления одних и тех же данных на клиенте и сервере. Универсального решения для этой проблемы не существует, поскольку без привлечения эксперта часто невозможно определить, какая версия данных является актуальной.

Решение поставленной задачи традиционно имеет несколько подходов (или их совокупность):

- установка приоритета клиента или сервера (кто окажется прав в спорных ситуациях);
- учет временной отметки каждой модификации данных (транзакции) — кто последний, тот прав;

- использование аддитивного метода — подходит для изменения данных, основанных на предыдущих значениях полей, например $salary = salary + 100$, в этом случае клиентское и серверное изменение можно обработать последовательно;
- разрешение конфликтов вручную. Ведется журнал всех конфликтов, и их разрешением занимается пользователь приложения.

В зависимости от специфики разрабатываемого приложения могут быть применены соответствующие подходы к разрешению конфликтов.

Замечание 3. Поскольку действия пользователя могут основываться на хранимой информации, обновление значений атрибутов объектов в локальной базе данных может существенно влиять на поведение пользователя. Таким образом, даже без появления конфликтов обновления данных возможны ситуации, когда пользовательские действия, выгружаемые из клиентского приложения на сервер посредством отложенных операций из очереди, будут противоречить состоянию базы данных сервера. Решение этой проблемы требует более внимательного рассмотрения изменений, произошедших на сервере с момента последнего сеанса связи.

6. Результаты

В работе предложена модель построения клиентского приложения, осуществляющего доступ к базе данных на сервере посредством API на основе CRUD-операций. Предложенная модель и алгоритм управляемой синхронизации дают возможность работать клиентскому приложению не только в режиме онлайн, но и при разрыве соединения с сервером. Отложенная согласованность данных достигается выполнением сохраненной на клиенте очереди вызовов методов серверного API, которая проходит дополнительное сжатие для оптимизации использования ресурсов на клиенте. В условиях невозможности изменения серверной части клиент-серверного решения предложенная архитектура позволяет реализовать приложение, сравнимое по возможностям с информационными системами, изначально разрабатываемыми с целью последующей синхронизации данных на клиенте и сервере.

Характерной особенностью предложенной модели является динамический учет действий, выполняемых клиентским приложением, заключающийся в хранении на стороне клиентского приложения только той информации, которая была запрошена на сервере. С другой стороны, если пользователь перестает пользоваться некоторыми функциями приложения, то соответствующие объекты базы данных не обновляются в клиентской базе данных, а могут быть и вовсе удалены. Таким образом, осуществляется упрощенная частичная синхронизация данных, определяемая поведением пользователя, а не полная синхронизация данных, при которой отслеживаются все изменения в обеих БД для проведения последующей синхронизации.

Основные особенности предложенного решения:

- решение не требует изменения программного кода на стороне сервера;
- предполагается динамическое определение нужд пользователя — одно и то же клиентское приложение, используемое разными людьми, может оперировать разными информационными блоками данных;
- возможна расстановка различных приоритетов атрибутам объектов с точки зрения важности их обновления на клиенте в контексте предметной области приложения;

- синхронизация локальной базы данных клиента с сервером является упрощенной и происходит лишь для части объектов, которые запрашиваются пользователем клиентского приложения.

Важно отметить, что использование предлагаемой модели помимо обеспечения возможности работы без подключения к серверу также может увеличить скорость работы приложений, отображая данные локальной БД, пока осуществляется удаленное выполнение соответствующей команды. Этот прием позволяет "сгладить" пользователям работу в приложении в условиях невысокой скорости подключения к сети Интернет.

Список литературы / References

[1] **Mulloy, B.** API facade pattern — a simple interface to a complex system. Available at: https://pages.apigee.com/api-facade-pattern-ebook.html (accessed 17.02.2016).

[2] **Gamma, E., Helm, R., Johnson, R.** Design patterns: Elements of reusable object-oriented software. Addison-Wesley, 1994. 395 p.

[3] **Subbu, A.** RESTful Web Services Cookbook. O'Reilly Media / Yahoo Press, 2010. 316 p.

[4] **Richardson, L., Amundsen, M., Ruby, S.** RESTful Web APIs. O'Reilly Media, 2013. 406 p.

[5] **Brewer, E.A.** Towards robust distributed systems // Proc. of the XIX Annual ACM Symp. on Principles of Distributed Computing. Portland, OR, 2000. 7 p.

[6] Документация по 1С-Битрикс: Управление сайтом. Адрес доступа: https://dev.1c-bitrix.ru/docs/php.php (дата обращения 17.02.2016).
Bitrix Site Manager documentation. Available at: https://dev.1c-bitrix.ru/docs/php.php (accessed 17.02.2016). (In Russ.)

[7] **Sen, S., Chatterjee, S., Dumir N.** Towards a theory of cache-efficient algorithms // J. of the ACM. 2002. Vol. 49(6). P. 828–858.

[8] **Coulouris, G., Dollimore, J., Kindberg T.** Distributed systems: Concepts and DESign. Addison-Wesley, 2007. 927 p.

[9] Evernote corporation evernote synchronization via EDAM. 2013. 15 p. Available at: https://dev.evernote.com/media/pdf/edam-sync.pdf (accessed 08.08.2016).

Поступила в редакцию 10 мая 2016 г.,
с доработки — 25 мая 2016 г.

Increase of resistance to breaks of network connections in client applications

Demish, Vsevolod O.[1,2], Pishchik, Boris N.[1,2,*]

[1]Design Technological Institute of Digital Techniques SB RAS, Novosibirsk, 630090, Russia
[2]Novosibirsk State University, Novosibirsk, 630090, Russia

Corresponding author: Pishchik, Boris N., e-mail: boris.pishchik@kti.nsc.ru

The main disadvantage of the client applications working with web services is the inability to get access to information when the connection is failed.

This paper introduces a model of the client application that accesses the database on the server via the API based on CRUD operations. The model assumes storage on the client computer as the local part of the database used and delayed synchronization. Data consistency on the client and a server is achieved by executing queue of method calls of he server side using API stored on the client side. Queue allows transformations to optimize resource usage for the client.

Only information that is requested on the server side is stored on the side of the client application. If the user ceases to use some functions of the application, then the corresponding objects of the database are not updated in the client database, and can be removed at all. Thus, we have the synchronization of data determined by the user behavior rather then the entire synchronization of the data.

The main features of the proposed solution:

- The solution does not require code changes on the server side.
- Dynamic control for the user's profile. The same client application, used by different people can handle different sets of information data.
- The ability of prioritizing the attributes of objects by the importance of their updates on the client side in the context of this domain.
- The simplified synchronization for the local client's database with the server. It is done only for a part of the objects that are requested by the client application.

Keywords: client server application, stability of the client applications, network disconnection.

Received 10 May 2016
Received in revised form 25 May 2016