Arenadata™ Streaming

Версия - v1.5-RUS

Руководство пользователя по работе с ADS

Оглавление

1 Руководство пользователя по работе с Nifi **3**
1.1 Обзор . . . 3
1.2 Поддерживаемые браузеры . . . 3
1.3 Терминология . . . 4
1.4 Пользовательский интерфейс NiFi . . . 6
1.5 Построение потока данных . . . 7
1.6 Настройка Процессора . . . 19
1.7 Пользовательские свойства с использованием Expression Language . . . 30
1.8 Управление потоком данных . . . 47

2 Руководство разработчика приложений для сервиса Kafka **50**
2.1 Kafka Clients . . . 50
2.2 Kafka Java Consumer . . . 54
2.3 Kafka Java Producer . . . 78
2.4 Kafka Connect . . . 84

В документации приведены сведения, которые могут быть полезны разработчикам приложений под Kafka и NiFi.

Инструкция полезна администраторам, программистам, разработчикам и сотрудникам подразделений информационных технологий, осуществляющих внедрение и сопровождение системы.

Important: Контактная информация службы поддержки – e-mail: info@arenadata.io

Глава 1

Руководство пользователя по работе с Nifi

В руководстве приведены сведения для пользователей системы по работе с платформой ADS в части серсиса Nifi – поддержка браузеров, особенности терминалогии, описание пользовательского интерфейса, создание потоков данных, настройка процессора, пользовательские свойства, а так же описаны команды и принцип управления потоками данных.

Руководство может быть полезно администраторам, программистам, разработчикам и сотрудникам подразделений информационных технологий, осуществляющих сопровождение платформы.

Important: Контактная информация службы поддержки – e-mail: info@arenadata.io

1.1 Обзор

Сервис **NiFi** – это система потоков данных, основанная на концепциях потокового программирования, которая поддерживает мощные и масштабируемые направленные графы маршрутизации данных, их преобразование и логичную системную медиацию. **NiFi** имеет веб-интерфейс пользователя для работы с потоками данных по их проектированию, управлению, обратной связи и мониторингу.

Сервис может быть сконфигурирован по нескольким параметрам качества обслуживания, таким как устойчивость к потерям и гарантированная доставка, низкая латентность и высокая пропускная способность, а также очередность на основе приоритетов. **NiFi** обеспечивает точное происхождение всех данных: полученных, forked-данных, клонированных при соединении, измененных, отправленных и, наконец, отброшенных при достижении своего настроенного конечного состояния.

Информация о системных требованиях, установке и настройке приведена в документе Руководство администратора по работе с сервисом Nifi.

Important: После установки сервиса NiFi необходимо использовать поддерживаемый веб-браузер для просмотра пользовательского интерфейса

1.2 Поддерживаемые браузеры

Сервис **Nifi** поддерживает работу с браузерами, представленными в таблице.

Таблица1.1.: Поддерживаемые браузеры

Браузер	Версия
Chrome	Актуальная и актуальная-1
FireFox	Актуальная и актуальная-1
Edge	Актуальная и актуальная-1
Safari	Актуальная и актуальная-1

Актуальная версия и актуальная-1 означает, что пользовательский интерфейс поддерживается в текущей стабильной версии данного браузера и в его предыдущей версии. Например, если текущий стабильный релиз браузера – *45.X*, то официально поддерживаемыми версиями являются *45.X* и *44.X*.

Для Safari, который выпускает основные версии гораздо реже других браузеров, актуальная версия и актуальная-1 представляют собой два последних выпуска.

Поддерживаемые версии браузера обусловлены возможностями пользовательского интерфейса и используемыми зависимостями, функции UI тестируются в указанных версиях браузеров. При этом пользовательский интерфейс может успешно работать в неподдерживаемых браузерах, но на них не проводится тестирование, и результат работы в них неизвестен. Кроме того, UI разработан как рабочий стол и в настоящее время не поддерживается в мобильных версиях браузеров.

В большинстве сред весь пользовательский интерфейс отображается в браузере. Тем не менее, UI имеет адаптивный дизайн, который позволяет прокручивать экраны по мере необходимости в браузерах меньшего размера или на планшетах. В средах, где ширина браузера меньше *800* пикселей, а высота меньше *600* пикселей, части пользовательского интерфейса могут быть недоступными.

1.3 Терминология

В руководстве используется характерная терминология, описание основных понятий приведено в далее.

DataFlow Manager, **DFM** – пользователь **NiFi**, имеющий разрешения на добавление, удаление и изменение компонентов потока данных **NiFi**.

FlowFile – файл потока – представляет собой единый фрагмент данных в **NiFi**, состоящий из двух компонентов: *FlowFile Attributes* – атрибуты и *FlowFile Content* – содержимое. Содержимое – это данные, которые представлены файлом потока. Атрибуты – это характеристики, которые предоставляют информацию или контекст о данных; они состоят из пар ключ-значение. Все FlowFiles имеют следующие стандартные атрибуты:

- *uuid* – уникальный идентификатор для FlowFile;
- *filename* – читаемое человеком имя файла, которое может использоваться при хранении данных на диске или на внешнем сервисе;
- *path* – иерархически структурированное значение, которое можно использовать при хранении данных на диске или на внешнем сервисе с целью, чтобы данные не хранились в одном каталоге.

Processor – процессор – это компонент **NiFi**, используемый для прослушивания входящих данных, извлечения данных из внешних источников, публикации данных во внешних источниках и для маршрутизации, преобразования и извлечения информации из FlowFiles.

Relationship – связь. Каждый процессор имеет ноль или более определенных для него связей. Связи указывают результат обработки файла потока FlowFile. После того как процессор завершает обработку FlowFile, он направляет его в одну из связей. Тогда DFM подключает каждую из связей к другим компонентам для определения, куда файл потока должен пойти далее после каждого потенциального результа обработки.

Connection – соединение. DFM создает автоматизированный поток данных, перемещая компоненты из панели инструментов “Components” на рабочую область, а затем соединяя компоненты через “Connections”.

Каждое соединение состоит из одной или нескольких связей. Для каждого проведенного соединения DFM может определить, какие связи следует использовать, что позволяет маршрутизировать данные различными способами в зависимости от результата обработки. Каждое соединение содержит очередь FlowFile и, когда FlowFile переносится в конкретную связь, он добавляется в принадлежащую соответствующему соединению очередь.

Controller Service – контроллер – это точки расширения, которые после добавления и настройки DFM в пользовательском интерфейсе запускаются при старте **NiFi** и предоставляют другим компонентам информацию для использования (например, процессорам или другими контроллерам). Общим контроллером, используемым несколькими компонентами, является *StandardSSLContextService*. Он предоставляет возможность конфигурировать свойства хранилища ключей и доверительного хранилища один раз, а затем повторно использовать эту конфигурацию во всем приложении. Идея заключается в том, что вместо того, чтобы настраивать данную информацию на каждом процессоре, который может нуждаться в ней, контроллер предоставляет ее любому процессору для использования по мере необходимости.

Reporting Task – задача отчетности. Задачи отчетности выполняются в фоновом режиме для предоставления статистических отчетов о происходящем в инстансе **NiFi**. DFM добавляет и настраивает Reporting Task в пользовательском интерфейсе по желанию. Общими задачами отчетности являются *ControllerStatusReportingTask*, *MonitorDiskUsage*, *MonitorMemory* и *StandardGangliaReporter*.

Funnel – воронка – это компонент **NiFi**, используемый для объединения данных из нескольких соединений в одно.

Process Group – группа процессов. Когда поток данных становится сложным, часто полезно рассмотреть его на более абстрактном уровне. **NiFi** позволяет группировать несколько компонентов, таких как процессоры, в группу процессов. При этом пользовательский интерфейс упрощает работу DFM для соединения нескольких Process Group в логический поток данных, а также позволяет DFM выводить группу процессов для просмотра и управления компонентами в ней.

Port – порт. Потокам данных, созданным с использованием одной или нескольких групп процессов, требуется способ подключения группы процессов к другим компонентам потока данных, что достигается за счет портов. DFM может добавить любое количество входных и выходных портов в группу процессов и присвоить им соответствующие имена.

Remote Process Group (RPG) – группа удаленных процессов. Так же, как данные перемещаются в группу процессов и из нее, порой необходимо перенести данные из одного инстанса **NiFi** в другой. В то время как **NiFi** предоставляет множество различных механизмов для передачи данных из одной системы в другую, удаленные группы процессов часто являются самым простым способом для достижения этой цели.

Bulletin – бюллетень. Пользовательский интерфейс **NiFi** выдает значительный объем мониторинга и обратной связи о текущем состоянии приложения. В дополнение к статистике и текущему статусу о каждом компоненте, компоненты могут сообщать бюллетени. Всякий раз, когда компонент сообщает бюллетень, на нем отображается соответствующий значок. На системном уровне бюллетени отображаются в строке состояния в верхней части страницы. Наведение указателя мыши на значок дает подсказку, показывающую время и критичность бюллетеня (*Debug*, *Info*, *Warning*, *Error*), а также само сообщение. Кроме того, бюллетени со всех компонентов можно просмотреть и отфильтровать на странице “Board Page” в общем меню.

Template – шаблон. Часто поток данных состоит из множества подпотоков, которые могут быть повторно использованы. **NiFi** позволяет DFM выбирать часть потока данных (или весь поток) и создавать шаблон. Шаблону присваивается имя и его можно переместить в рабочую область так же, как и другие компоненты. В результате несколько компонентов могут быть объединены вместе, чтобы сделать более крупный блок для создания потока данных. Шаблоны также можно экспортировать как XML и импортировать в другой инстанс **NiFi**, обеспечивая совместное использование.

flow.xml.gz – все, что DFM помещает в рабочую область пользовательского интерфейса **NiFi**, записывается в реальном времени в один файл с именем *flow.xml.gz*. По умолчанию файл находится в каталоге *nifi/conf*. Любые изменения, сделанные в рабочей области, автоматически сохраняются без необходимости принудительного сохранения посредством кнопки “Save”. Кроме того, **NiFi** автоматически создает резервную

копию данного файла в каталоге архива при его обновлении, которую можно использовать для настройки отката потока. Для этого необходимо остановить **NiFi**, заменить *flow.xml.gz* на требуемую резервную копию и перезапустить **NiFi**. В кластерной среде следует остановить весь кластер **NiFi**, заменить *flow.xml.gz* одного из узлов и перезапустить данный узел. Затем удалить *flow.xml.gz* из других узлов. После подтверждения, что узел запускается как кластер с одним узлом, запустить остальные узлы. При этом замененная конфигурация потока синхронизируется по всему кластеру. Имя, расположение *flow.xml.gz* и поведение автоматической архивации настраиваемы.

1.4 Пользовательский интерфейс NiFi

Пользовательский интерфейс (UI) **NiFi** предоставляет механизмы для создания автоматизированных потоков данных, а также их визуализации, редактирования, мониторинга и администрирования. Пользовательский интерфейс может быть разбит на несколько сегментов, отвечающих за различные функции приложения. Далее представлены некоторые скриншоты с выделенными сегментами UI приложения с подробным описанием основных моментов.

При запуске приложения пользователь может перейти к пользовательскому интерфейсу, выбрав в веб-браузере адрес по умолчанию *http://<hostname>:8080/nifi*. Так как изначально разрешения не настроены, любой пользователь может просматривать и изменять поток данных. Конфигурация настроек приведена в Руководстве администратора по работе с сервисом Nifi.

При первом переходе DFM к пользовательскому интерфейсу отображается пустая рабочая область (Рис.1.1.).

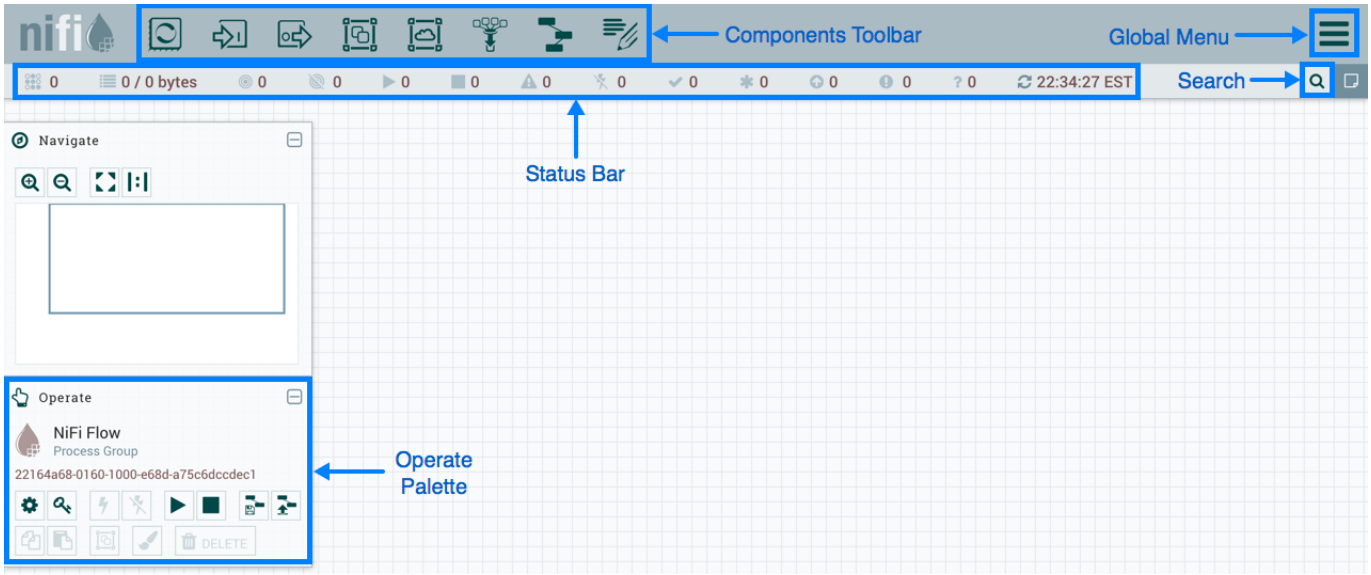

Рис.1.1.: Рабочая область NiFi

Панель инструментов “Components Toolbar” располагается в левой верхней части экрана. Она состоит из компонентов, которые можно переносить в рабочую область для построения потока данных. Каждый компонент панели описан в главе Построение потока данных.

Строка состояния “Status Bar” находится под панелью “Components Toolbar”. Она предоставляет информацию о количестве активных потоков на текущий момент времени, об объеме данных в настоящее время в потоке, количестве групп удаленных процессов в рабочей области в каждом состоянии (*Transmitting*, *Not Transmitting*), сколько процессоров существует в рабочей области в каждом состоянии (*Stopped*, *Running*, *Invalid*, *Disabled*), сколько версий групп процессов существует в рабочей области в каждом состоянии (*Up to date*, *Locally modified*, *Stale*, *Locally modified and stale*, *Sync failure*) и временная метка, в которой вся приведенная

информация обновлена в последний раз. Кроме того, если инстанс **NiFi** кластеризован, в строке состояния отображается количество узлов в кластере и сколько из них в настоящее время подключено.

Палитра “Operate” находится в левой части экрана. Она состоит из кнопок управления потоком для DFM, а также для администраторов, управляющих доступом пользователей и настраивающих системные свойства, например, количество предоставленных приложению системных ресурсов.

В правой части рабочей области находится поиск и общее меню. С помощью поиска можно легко находить компоненты в рабочей области по их имени, типу, идентификатору, свойствам конфигурации и их значениям. Общее меню содержит параметры, позволяющие управлять существующими компонентами (Рис.1.2.).

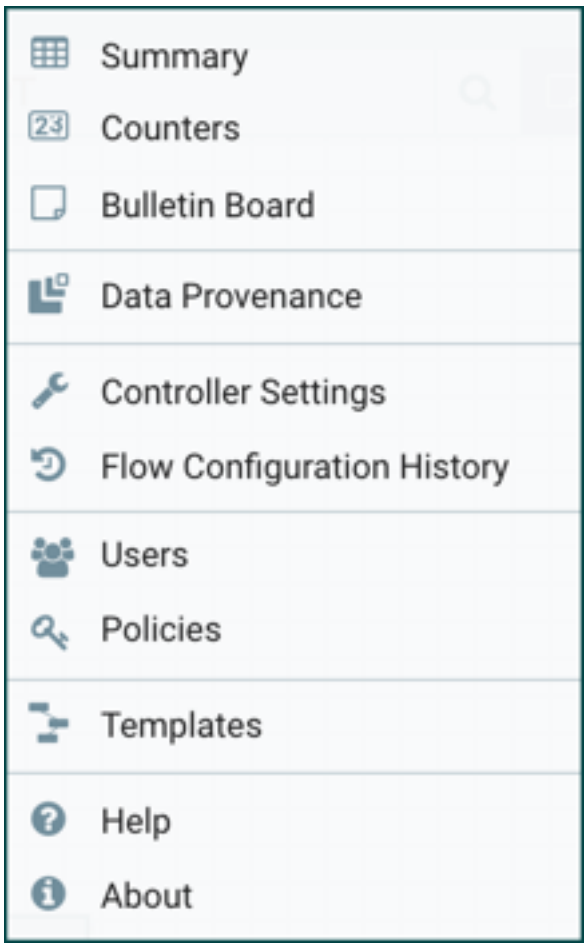

Рис.1.2.: Пармаметры общего меню NiFi

Кроме того, пользовательский интерфейс имеет панель навигации “Navigate”, позволяющую легко перемещаться по рабочей области с возможностью увеличения и уменьшения масштаба. Отдаленный вид потока данных обеспечивает высокоуровневое представление потока и позволяет перемещаться между его крупными частями. Вдоль нижней части экрана располагается навигационная цепочка (“Breadcrumbs”). При переходе в группу процессов цепочка отображают глубину потока и его группы, каждая из которой в свою очередь представляет собой ссылку, по которой можно перейти на тот или иной уровень в потоке (Рис.1.3.).

1.5 Построение потока данных

Автоматизированный поток данных создается DFM через пользовательский интерфейс **NiFi**. Для этого необходимо перенести компоненты с панели инструментов на рабочую область, настроить их в соответствии с

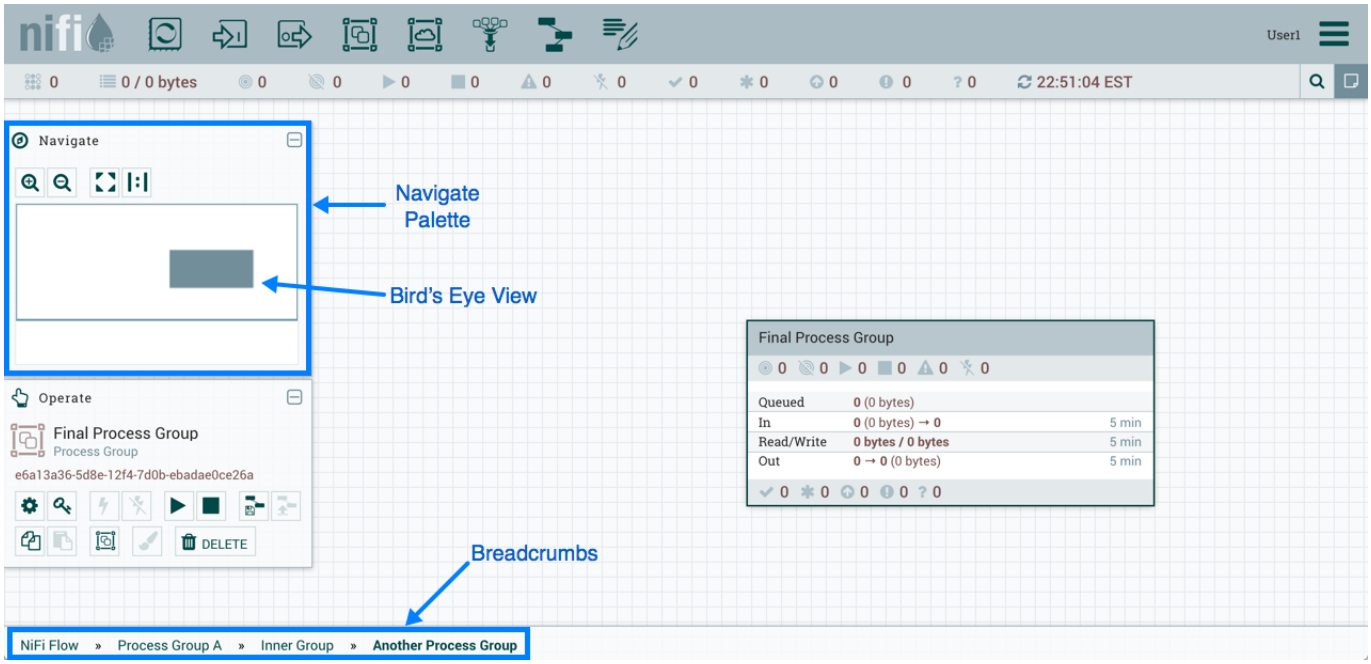

Рис.1.3.: Навигация в NiFi

конкретными потребностями и соединить вместе.

1.5.1 Добавление компонентов на рабочую область

В разделе Пользовательский интерфейс NiFi описаны различные сегменты UI и упомянута панель инструментов “Components Toolbar”, разбор компонентов которой приведен далее (Рис.1.4.).

Processor – Процессор является наиболее часто используемым компонентом, поскольку он отвечает за поступление и выход данных, их маршрутизацию и управление. Существуют различные типы Процессоров. Фактически, это очень распространенная точка расширения в **NiFi**, что означает, что поставщики могут реализовать свои собственные Процессоры для выполнения необходимых для их использования функций. При перемещении Процессора на рабочую область **NiFi** открывается диалоговое окно выбора типа (Рис.1.5.).

В правом верхнем углу можно задать фильтр списка на основе типа Процессора или связанных с ним тегов. Разработчики имеют возможность присваивания тегов к своим Процессорам, и в дальнейшем данные теги отображаются с левой стороны в специальном облаке. Щелчок по тегу в облаке отсортировывает только содержащие данный тег Процессоры. При выборе нескольких тегов отображаются только те Процессоры, которые содержат все указанные теги вкупе. Например, если необходимо показать только те Процессоры, которые позволяют загружать файлы, можно выбрать теги *files* и *ingest* (Рис.1.6.).

Ограниченные компоненты отмечаются соответствующим значком слева от названия их типа. Это компоненты, которые могут использоваться для выполнения произвольного необработанного кода, предоставляемого оператором через REST API/UI, или которые могут использоваться для получения или изменения данных в хост-системе **NiFi** с применением учетных данных ОС. Ограниченные компоненты также необходимы авторизованным пользователем **NiFi** с целью выхода за пределы предполагаемого использования приложения, повышения привилегий или предоставления информации о внутренних компонентах процесса **NiFi** или хост-системы.

Все эти возможности следует считать привилегированными, администраторам необходимо знать о них и явно включать для подмножества доверенных пользователей. Но прежде чем пользователь сможет создавать и изменять ограниченные компоненты, ему должен быть предоставлен доступ на определенные разрешения, которые отображаются при наведении курсора на значок “Restricted”. При этом разрешения могут назначаться

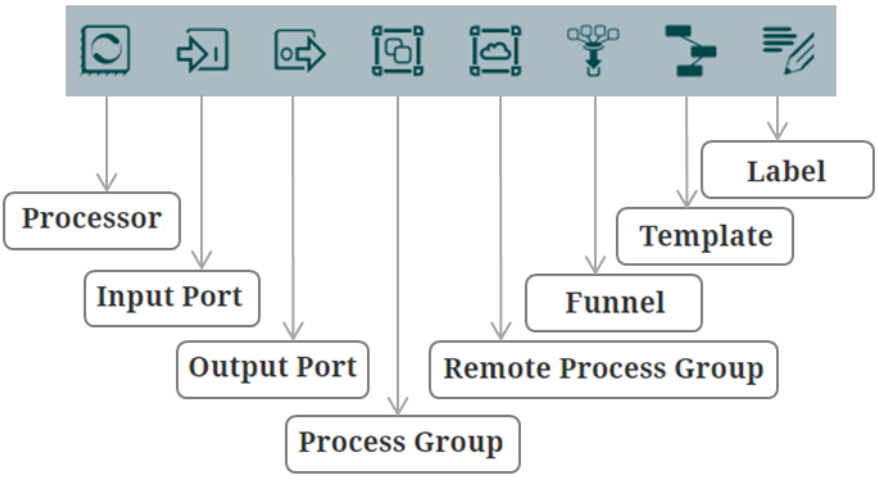

Рис.1.4.: Разбор панели инструментов NiFi

независимо от ограничений. В таком случае пользователь имеет доступ ко всем ограниченным компонентам. Также пользователям может быть назначен доступ к определенным ограничениям. И в случае если пользователю предоставляется доступ ко всем ограничениям, требуемым компоненту, то он будет иметь доступ к данному компоненту (при условии наличия достаточных разрешений).

Нажатие кнопки “Add” или двойное нажатие на тип Процессора добавляет выбранный Процессор в рабочую область на то место, куда была перемещена пиктограмма с панели инструментов.

Important: Любой компонент в рабочей области можно выбрать с помощью мыши и переместить. Также можно управлять несколькими элементами одновременно – удерживая нажатой клавишу Shift и выбрав каждый компонент

С Процессором в рабочей области можно взаимодействовать, щелкнув правой кнопкой мыши по нему и выбрав параметр из выпадающего контекстного меню. Доступные опции различаются в зависимости от назначенных пользователю привилегий (Рис.1.7.).

Хотя опции контекстного меню различаются, приведенные далее обычно доступны:

- *Configure* – опция позволяет пользователю устанавливать или изменять конфигурацию Процессора (см. Настройка Процессора). Для Процессоров, портов, групп удаленных процессов, соединений и меток диалоговое окно конфигурации можно открыть двойным щелчком по компоненту;
- *Start* или *Stop* – опция позволяет пользователю запускать или останавливать Процессор, в зависимости от его текущего состояния;
- *Enable* или *Disable* – опция позволяет пользователю включить или отключить Процессор, в зависимости от его текущего состояния;
- *View data provenance* – опция отображает таблицу происхождения данных *NiFi Data Prevenance* с информацией о событиях для FlowFiles, проложенных через выбранный Процессор;
- *View status history* – опция открывает графическое представление статистической информации Процессора с течением времени;
- *View usage* – опция позволяет перейти к документации по использованию Процессора;

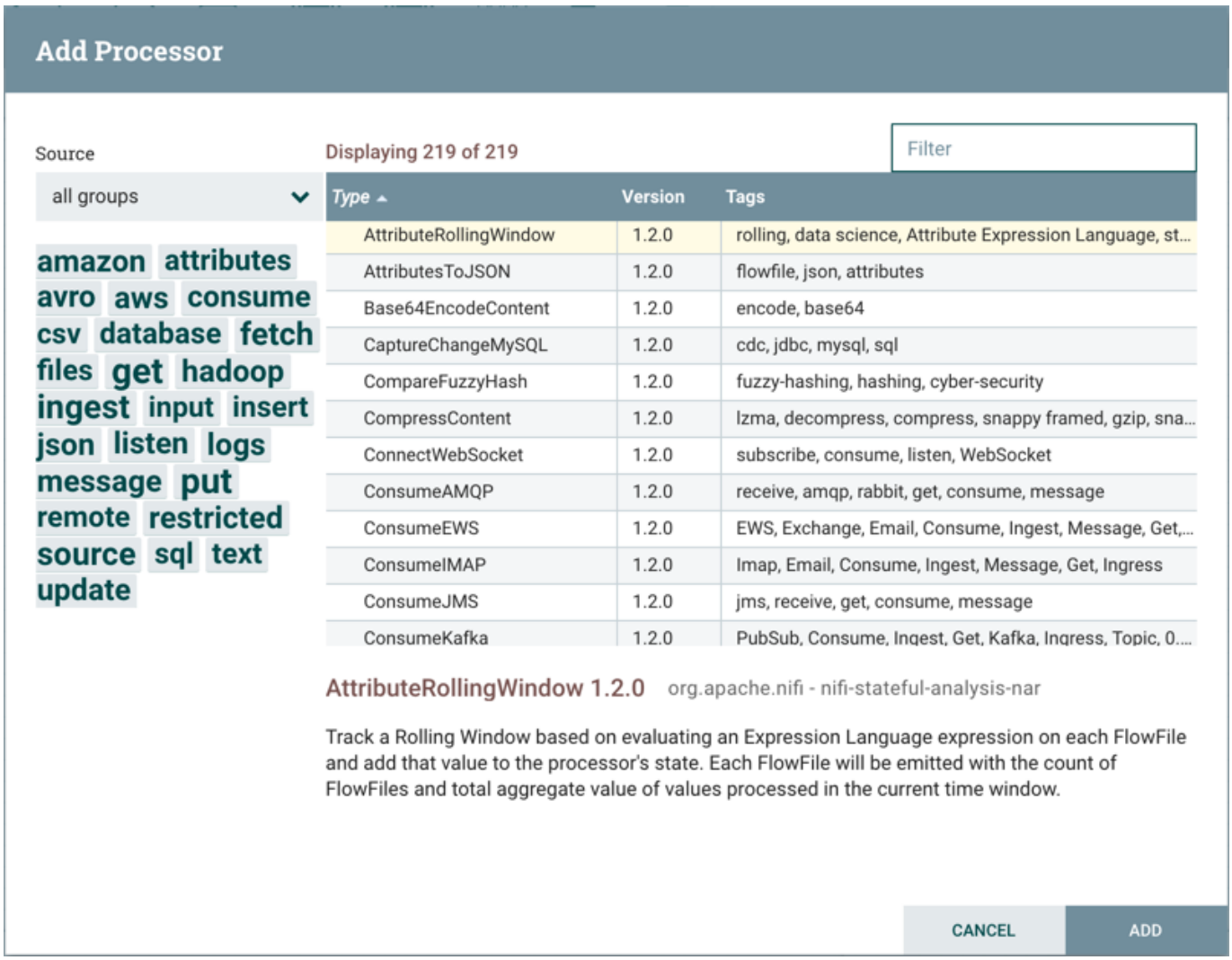

Рис.1.5.: Выбор типа Процессора

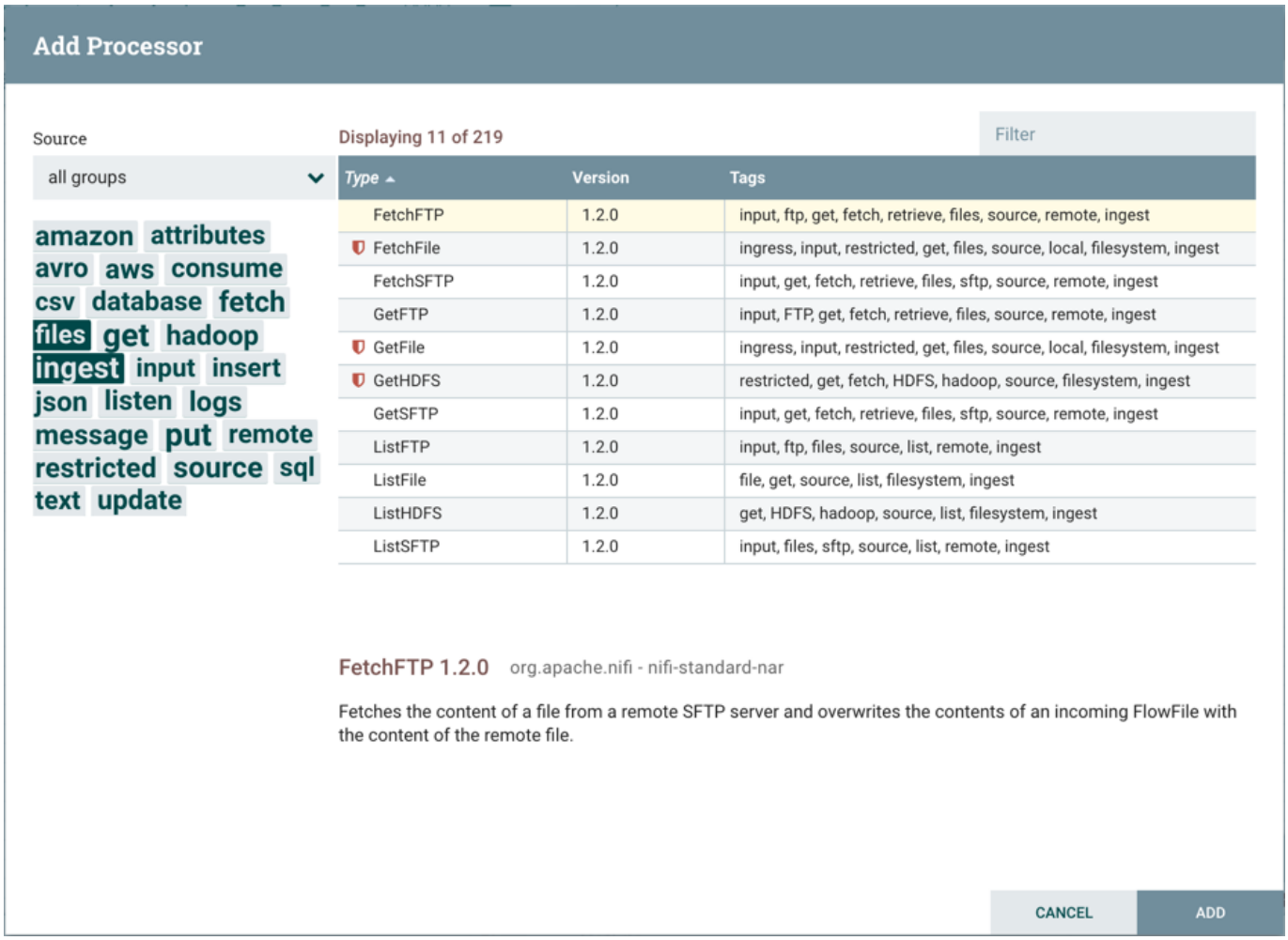

Рис.1.6.: Фильтрация Процессоров по тегам

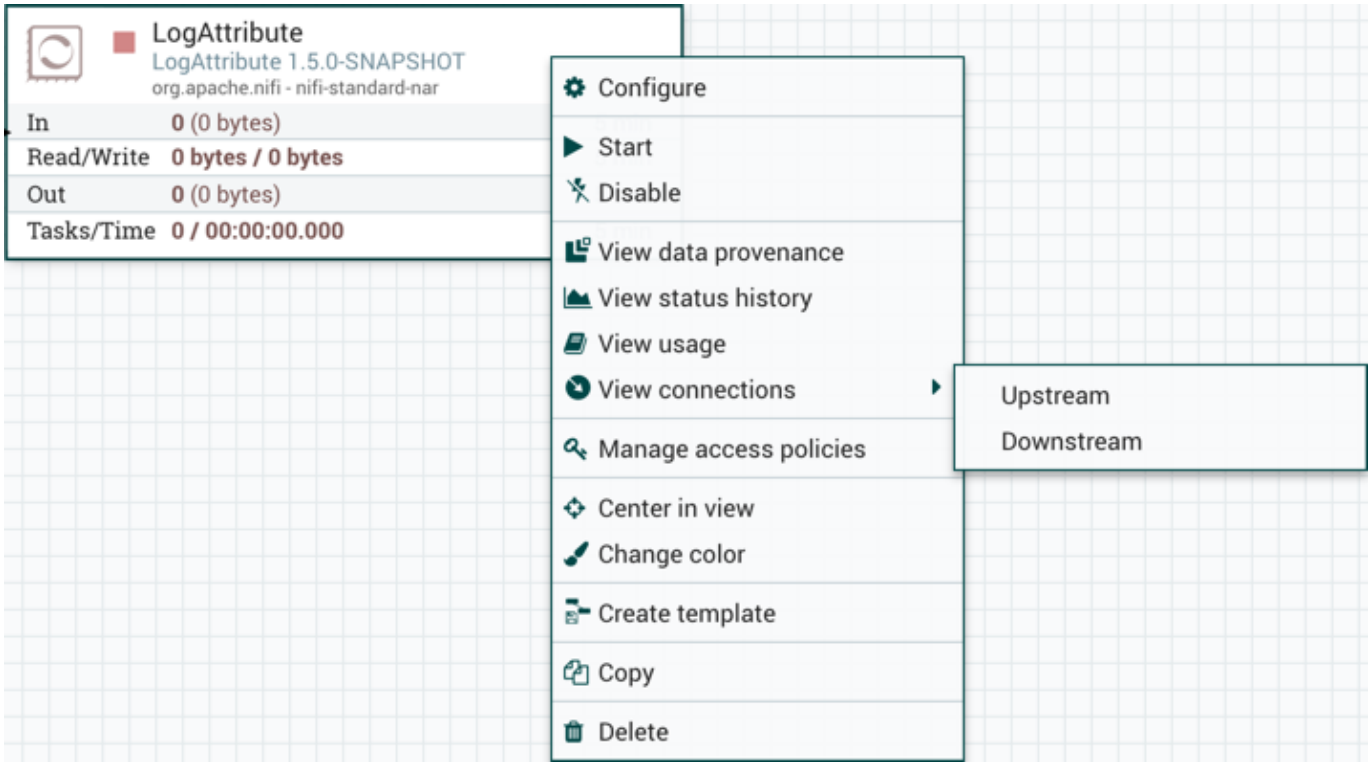

Рис.1.7.: Опции Процессора

- *View connections → Upstream* – опция позволяет пользователю видеть и переходить на восходящие соединения, входящие в Процессор. Это особенно полезно, когда Процессоры подключаются к другим группам процессов и выходят из них;
- *View connections → Downstream* – опция позволяет пользователю видеть и переходить на нисходящие соединения, входящие в Процессор. Это особенно полезно, когда Процессоры подключаются к другим группам процессов и выходят из них;
- *Center in view* – опция центрирует представление рабочей области на данном Процессоре;
- *Change color* – опция позволяет пользователю изменять цвет Процессора, что упрощает визуальный менеджмент больших потоков;
- *Create template* – опция позволяет пользователю создать шаблон из выбранного Процессора;
- *Copy* – опция помещает копию выбранного Процессора в буфер обмена, чтобы можно было его добавить в другое место рабочей области, щелкнув правой кнопкой мыши и выбрав “Paste”. Действия Copy/Paste также могут выполняться с помощью комбинации клавиш “Ctrl-C” (“Command-C”) и “Ctrl-V” (“Command-V”);
- *Delete* – опция позволяет DFM удалять Процессор с рабочей области.

Input Port – Входной порт предоставляет механизм для передачи данных в группу процессов. Когда входной порт перемещается на рабочую область, DFM получает запрос на имя порта. Все порты в группе процессов должны иметь уникальные имена.

Все компоненты существуют в Process Group. Когда пользователь изначально переходит на страницу **NiFi**, он помещается в Root Process Group. Если входной порт перемещается в данную группу процессов, входной порт обеспечивает механизм для приема данных из удаленных инстансов **NiFi** посредством Site-to-Site. В таком случае входной порт может быть настроен для ограничения доступа к соответствующим пользователям при

настройке безопасного запуска **NiFi**.

Output Port – Выходной порт предоставляет механизм для передачи данных из группы процессов в места назначения за ее пределами. Когда выходной порт перемещается на рабочую область, DFM получает запрос на имя порта. Все порты в группе процессов должны иметь уникальные имена.

Если выходной порт перемещается в Root Process Group, он обеспечивает механизм отправки данных удаленным инстансам **NiFi** посредством Site-to-Site. В таком случае порт действует как очередь. Поскольку удаленные инстансы **NiFi** извлекают данные из порта, эти данные удаляются из очередей входящих соединений. При настройке безопасного запуска **NiFi** выходной порт можно сконфигурировать для ограничения доступа к соответствующим пользователям.

Process Group – Группы процессов могут использоваться для логического объединения набора компонентов с целью упрощения понимания и управления потоком данных. Когда группа процессов перемещается на рабочую область, у DFM запрашивается имя Process Group, после чего группа процессов вкладывается в родительскую группу. Все Process Group в одной родительской группе должны иметь уникальные имена.

С группой процессов в рабочей области можно взаимодействовать, щелкнув правой кнопкой мыши по ней и выбрав параметр из выпадающего контекстного меню. Доступные опции различаются в зависимости от назначенных пользователю привилегий (Рис.1.8.).

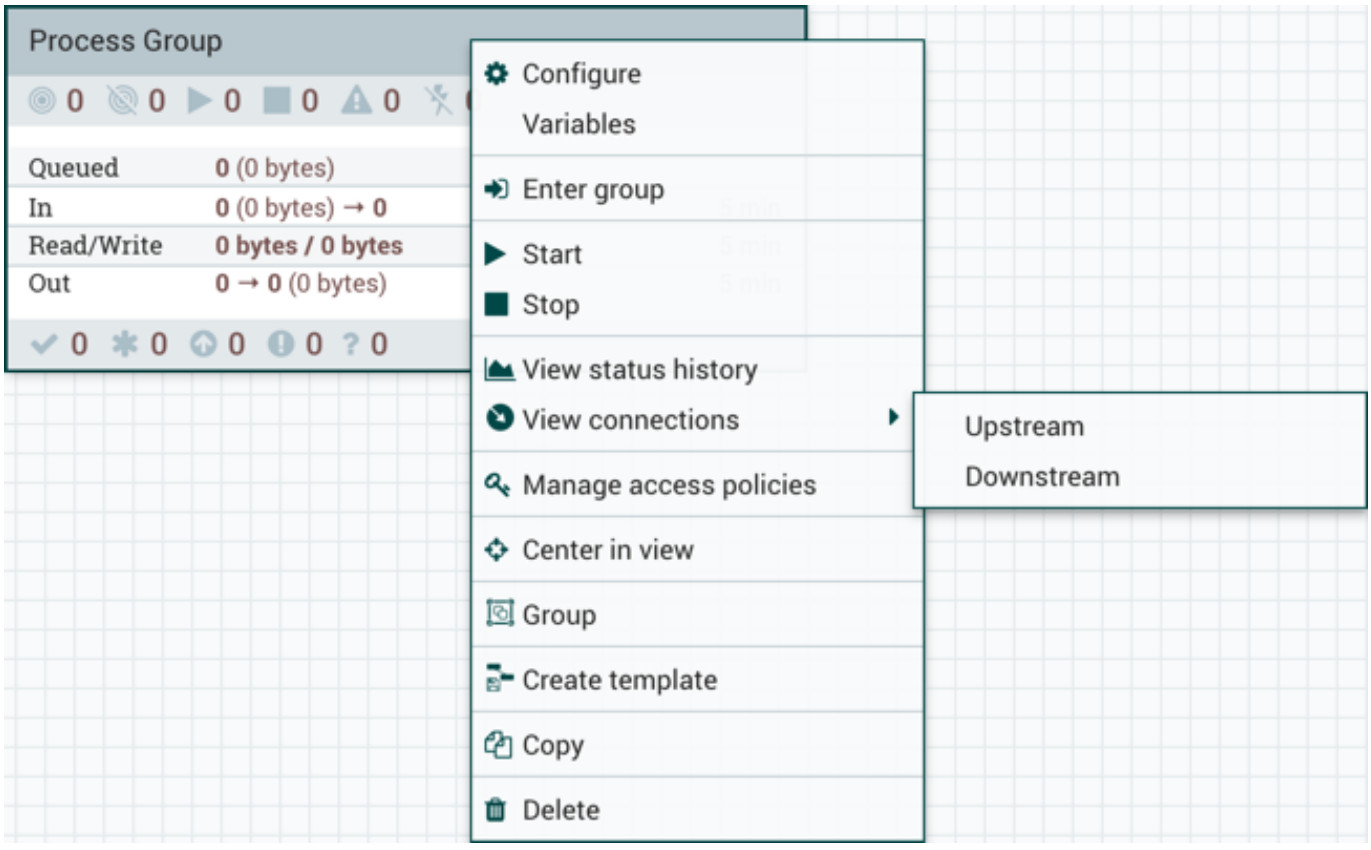

Рис.1.8.: Опции группы процессов

Хотя опции контекстного меню различаются, приведенные далее обычно доступны:

- *Configure* – опция позволяет пользователю устанавливать или изменять конфигурацию группы процессов;
- *Variables* – опция позволяет пользователю создавать или настраивать переменные в пользовательском интерфейсе **NiFi**;

- *Enter group* – опция позволяет пользователю войти в группу процессов. Также можно дважды щелкнуть по группе процессов, чтобы войти в нее;
- *Start* – опция позволяет пользователю запустить группу процессов;
- *Stop* – опция позволяет пользователю остановить группу процессов;
- *View status history* – опция открывает графическое представление статистической информации группы процессов с течением времени;
- *View connections* → *Upstream* – опция позволяет пользователю видеть и переходить на восходящие соединения, входящие в группу процессов;
- *View connections* → *Downstream* – опция позволяет пользователю видеть и переходить на нисходящие соединения, входящие в группу процессов;
- *Center in view* – опция центрирует представление рабочей области на данной группе процессов;
- *Group* – опция позволяет пользователю создать новую группу процессов, содержащую выбранную и любые другие компоненты, указанные на рабочей области;
- *Create template* – опция позволяет пользователю создать шаблон из выбранной группы процессов;
- *Copy* – опция помещает копию выбранной группы процессов в буфер обмена, чтобы можно было ее добавить в другое место рабочей области, щелкнув правой кнопкой мыши и выбрав “Paste”. Действия Copy/Paste также могут выполняться с помощью комбинации клавиш “Ctrl-C” (“Command-C”) и “Ctrl-V” (“Command-V”);
- *Delete* – опция позволяет DFM удалять группу процессов с рабочей области.

Remote Process Group – Группы удаленных процессов отображаются и ведут себя аналогично группам процессов. Только группа удаленных процессов (RPG) ссылается на удаленный инстанс **NiFi**. Когда RPG перемещается на рабочую область, у DFM запрашивается URL-адрес удаленного инстанса. Если удаленный **NiFi** является кластеризованным, URL-адрес, который должен использоваться, – это URL-адрес любого инстанса **NiFi** в этом кластере. Когда данные передаются кластеризованному **NiFi** через RPG, RPG подключается к удаленному инстансу, URL-адрес которого настроен для определения, какие узлы находятся в кластере и насколько занят каждый из них. Эта информация используется для балансировки загрузки данных на каждый узел. Затем удаленные инстансы периодически опрашиваются для определения сведений о узлах, которые удаляются из кластера или добавляются в него, и при этом балансировка загрузки каждого узла перерасчитывается.

С удаленной группой процессов в рабочей области можно взаимодействовать, щелкнув правой кнопкой мыши по ней и выбрав параметр из выпадающего контекстного меню. Доступные опции различаются в зависимости от назначенных пользователю привилегий (Рис.1.9.).

Хотя опции контекстного меню различаются, приведенные далее обычно доступны:

- *Configure* – опция позволяет пользователю устанавливать или изменять конфигурацию группы удаленных процессов;
- *Enable transmission* – опция активирует передачу данных между инстансами **NiFi**;
- *Disable transmission* – опция отключает передачу данных между инстансами **NiFi**;
- *View status history* – опция открывает графическое представление статистической информации группы удаленных процессов с течением времени;
- *View connections* → *Upstream* – опция позволяет пользователю видеть и переходить на восходящие соединения, входящие в группу удаленных процессов;
- *View connections* → *Downstream* – опция позволяет пользователю видеть и переходить на нисходящие соединения, входящие в группу удаленных процессов;
- *Refresh remote* – опция обновляет представление состояния удаленного инстанса **NiFi**;

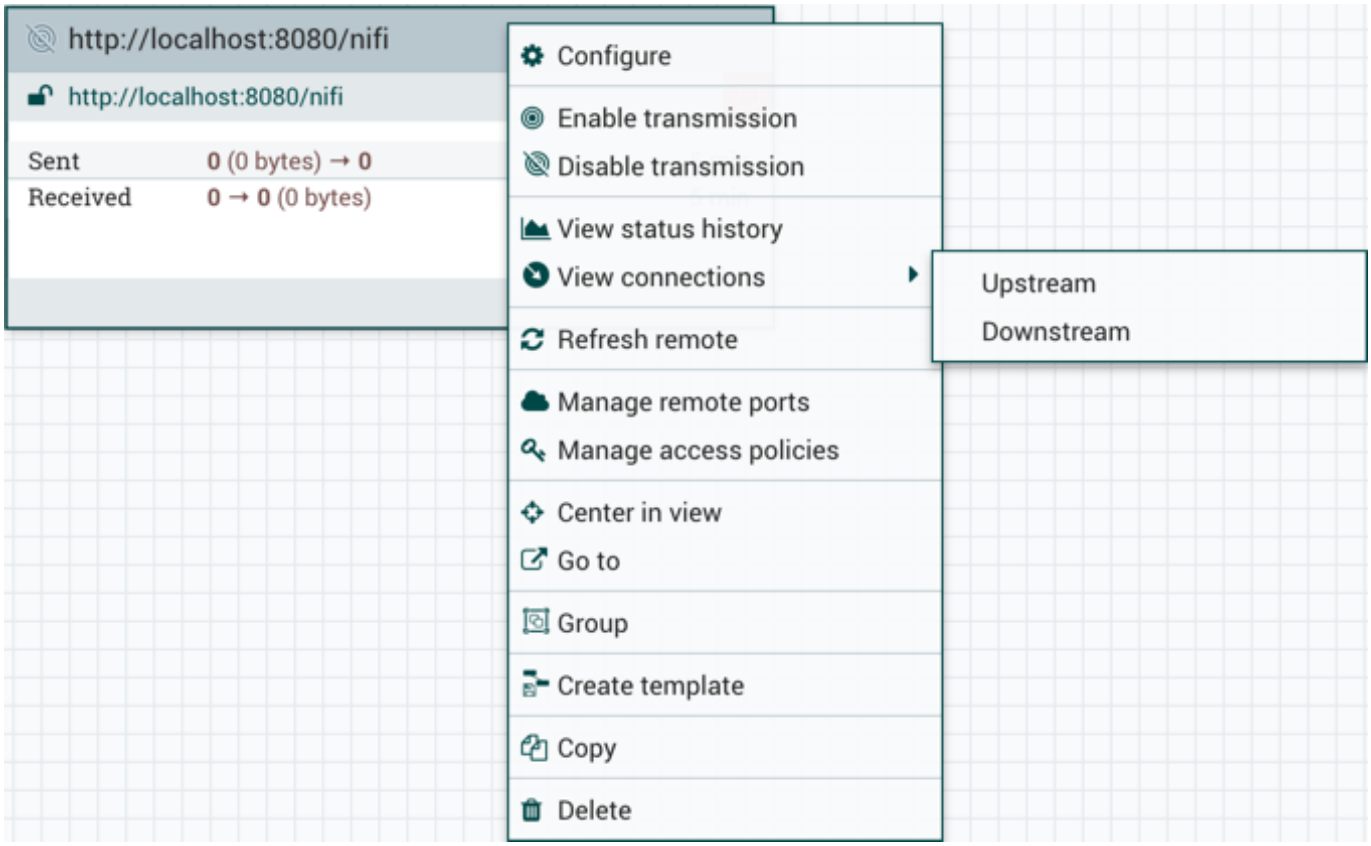

Рис.1.9.: Опции удаленной группы процессов

- *Manage remote ports* – опция позволяет пользователю видеть входные и/или выходные порты, существующие на удаленном инстансе **NiFi**, к которому подключена группа удаленных процессов. При этом если конфигурация Site-to-Site защищена, отображаются только те порты, к которым предоставлен доступ данному пользователю **NiFi**;
- *Center in view* – опция центрирует представление рабочей области на данной группе удаленных процессов;
- *Go to* – опция открывает представление удаленного инстанса **NiFi** на новой вкладке браузера. При этом если конфигурация Site-to-Site защищена, у пользователя должен быть доступ к удаленному инстансу **NiFi** для его просмотра;
- *Group* – опция позволяет пользователю создать группу процессов, содержащую выбранную группу удаленных процессов;
- *Create template* – опция позволяет пользователю создать шаблон из выбранной группы удаленных процессов;
- *Copy* – опция помещает копию выбранной группы удаленных процессов в буфер обмена, чтобы можно было ее добавить в другое место рабочей области, щелкнув правой кнопкой мыши и выбрав “Paste”. Действия Copy/Paste также могут выполняться с помощью комбинации клавиш “Ctrl-C” (“Command-C”) и “Ctrl-V” (“Command-V”);
- *Delete* – опция позволяет DFM удалять группу удаленных процессов с рабочей области.

Funnel – Воронки используются для объединения данных из нескольких Соединений в одно, что имеет два преимущества. Во-первых, при наличии большого количества Соединений с одним и тем же назначением рабочая область может загромождаться занимаемым ими пространством. Путем объединения Соединений в одно, полученное одиночное Соединение затем можно так же нарисовать на рабочей области, охватив такое же пространство. Во-вторых, Соединения могут быть настроены с помощью приоритетов FlowFile. Данные из нескольких Соединений могут быть направлены в одиночное Соединение, обеспечивая возможность приоритизации всех данных, а не определять приоритеты данных по каждому Соединению независимо друг от друга.

Template – Шаблоны могут создаваться DFM из части потока или могут импортироваться из других потоков данных. Они обеспечивают крупные блоки для быстрого создания сложного потока. При перемещении пиктограммы “Template” на рабочую область открывается диалоговое окно для выбора шаблона из списка доступных (Рис.1.10.).

В раскрывающемся списке находятся все доступные шаблоны. Любой шаблон, созданный с описанием, содержит значок вопроса, указывающий на наличие дополнительных сведений, отображающихся при наведении курсора мыши на иконку (Рис.1.11.).

Label – Ярлыки используются для предоставления информативного текста частям потока данных. При перемещении пиктограммы “Label” на рабочую область он создается с заданным по умолчанию размером с возможностью последующего редактирования при помощи маркера в правом нижнем углу. Ярлык не имеет текста при создании. Текст добавляется по щелчку правой кнопкой мыши на ярлыке и выбору параметра *Configure*.

1.5.2 Версии компонентов

В приложении есть доступ к информации о версии Процессоров, контроллера и задач отчетности. Это особенно полезно при работе в кластерной среде с несколькими инстансами **NiFi**, использующими разные версии компонентов, или при обновлении до более новой версии процессора. Диалоговые окна “Add Processor”, “Add Controller Service” и “Add Reporting Task” содержат столбец с версией компонента, а также имя компонента, организации или группы, создавшей его, и содержащий данный компонент пакет NAR (Рис.1.12.).

Каждый компонент на рабочей области также содержит эту информацию (Рис.1.13.).

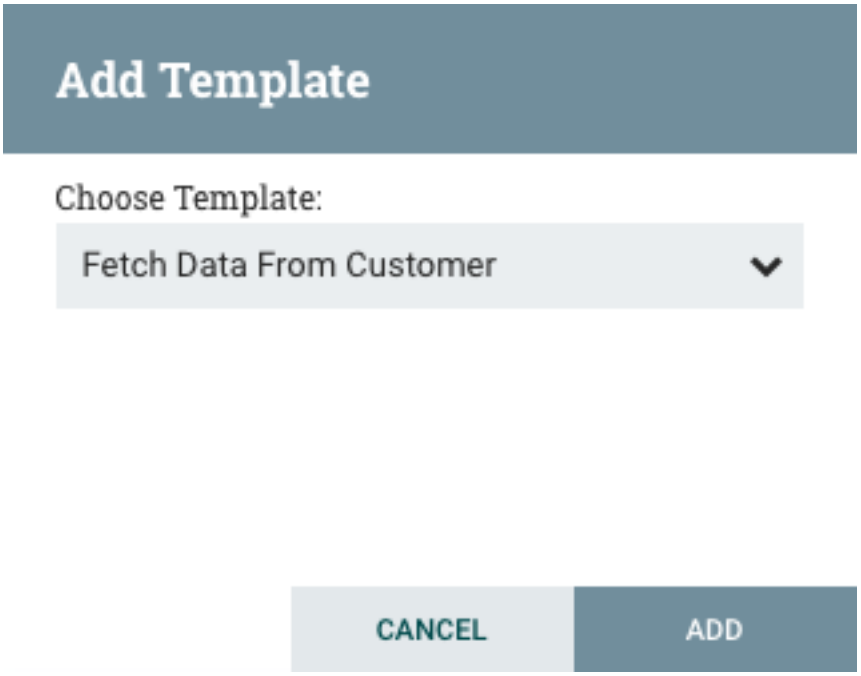

Рис.1.10.: Выбор шаблона

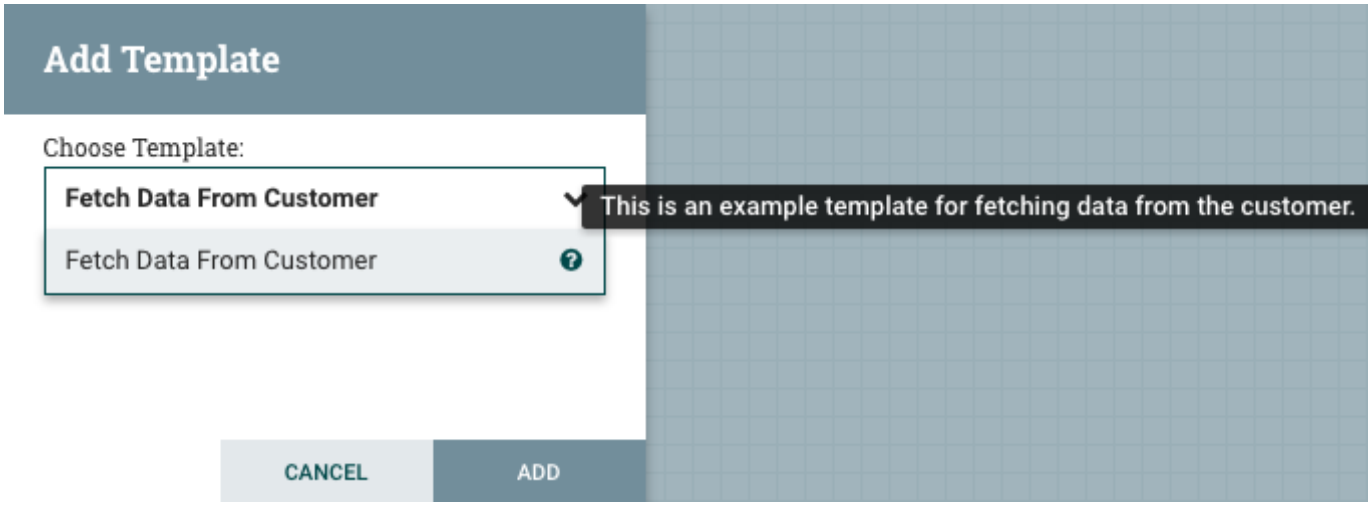

Рис.1.11.: Дополнительные сведения о шаблоне

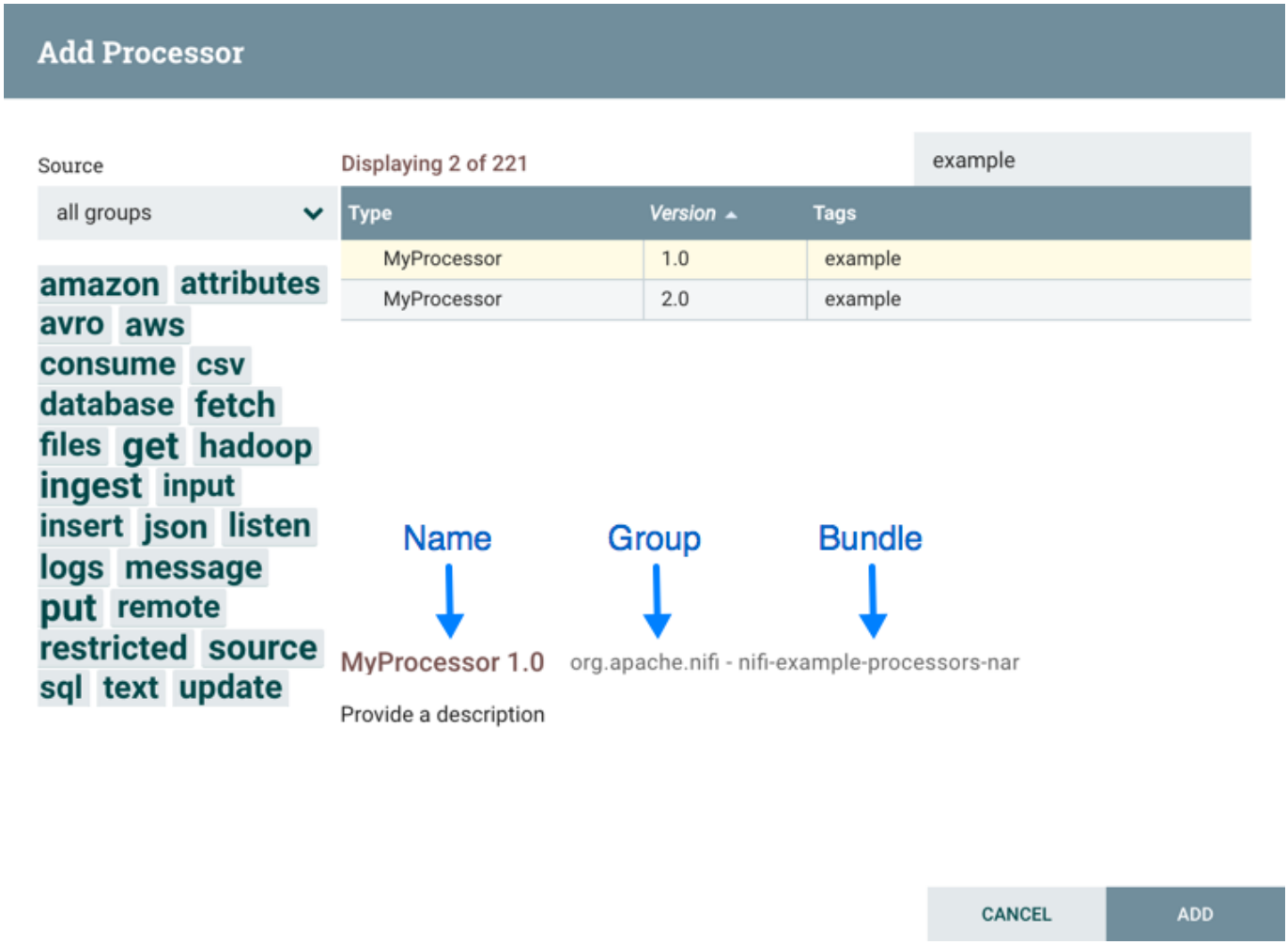

Рис.1.12.: Версии компонентов

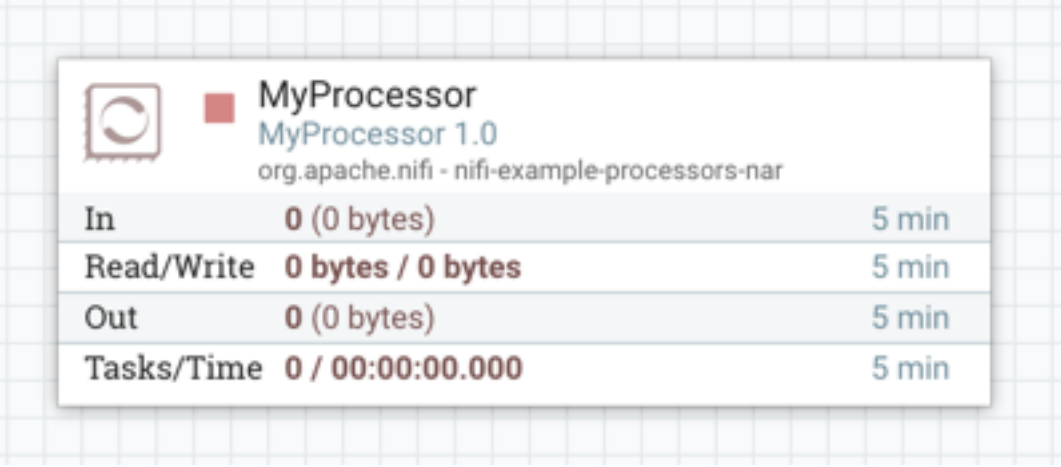

Рис.1.13.: Версия компонента

1.5.3 Изменение версии компонента

Для изменения версии компонента необходимо выполнить следующие действия:

1. Кликнуть правой кнопкой мыши на компонент в рабочей области для отображения параметров конфигурации.
2. Выбрать “Change version” (Рис.1.14.).

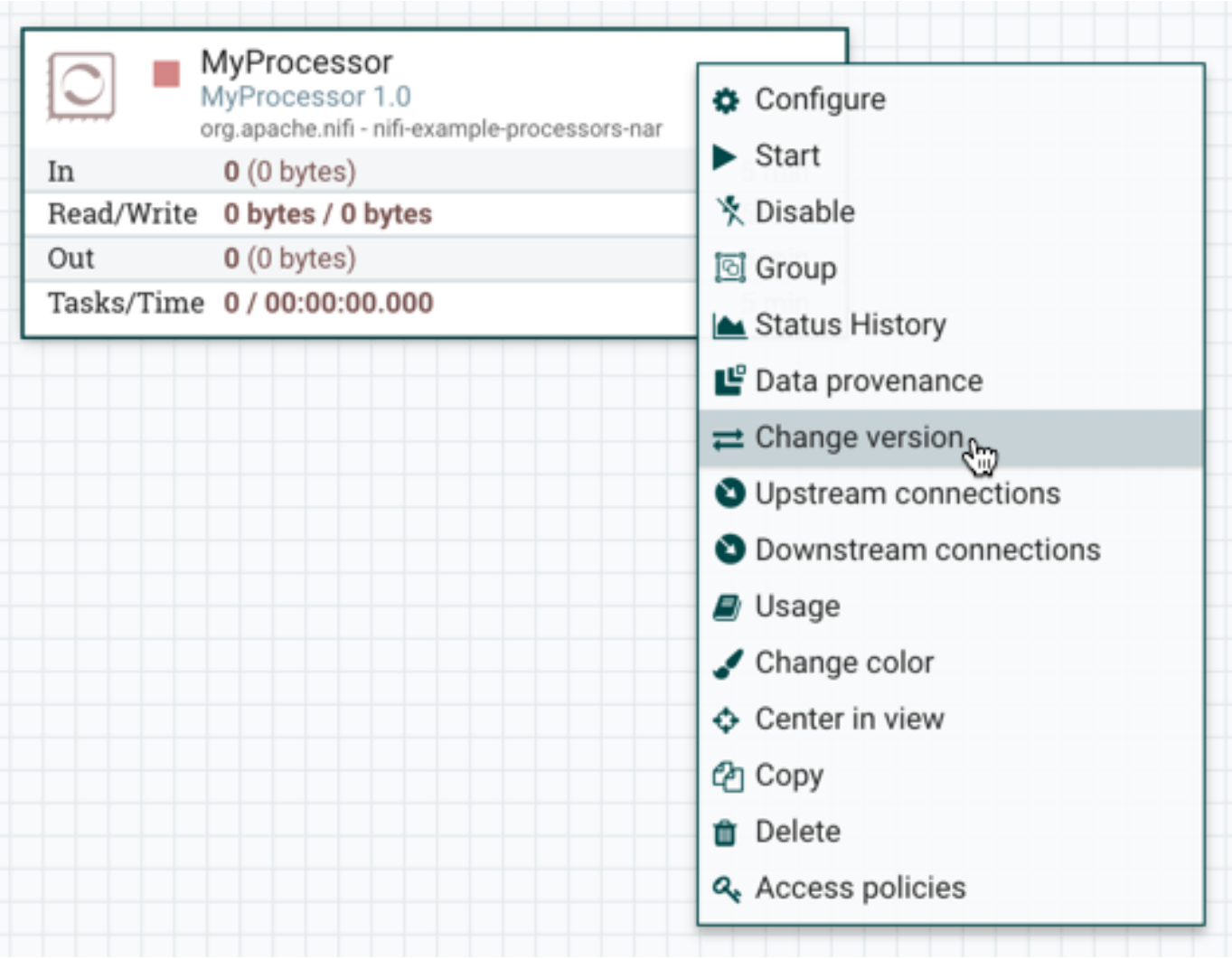

Рис.1.14.: Параметры конфигурации компонента

3. В диалоговом окне “Component Version” выбрать необходимую версию в раскрывающемся меню “Version” (Рис.1.15.).

1.6 Настройка Процессора

Окно настройки Процессора открывается по двойному щелчку по Процессору в рабочей области либо при выборе опции “Configure” из контекстного меню Процессора. Диалоговое окно конфигурации имеет четыре вкладки, каждая из которых описана далее. После завершения настройки Процессора для применения изменений следует нажать кнопку “Apply”; для выхода из окна без сохранения – кнопку “Cancel”.

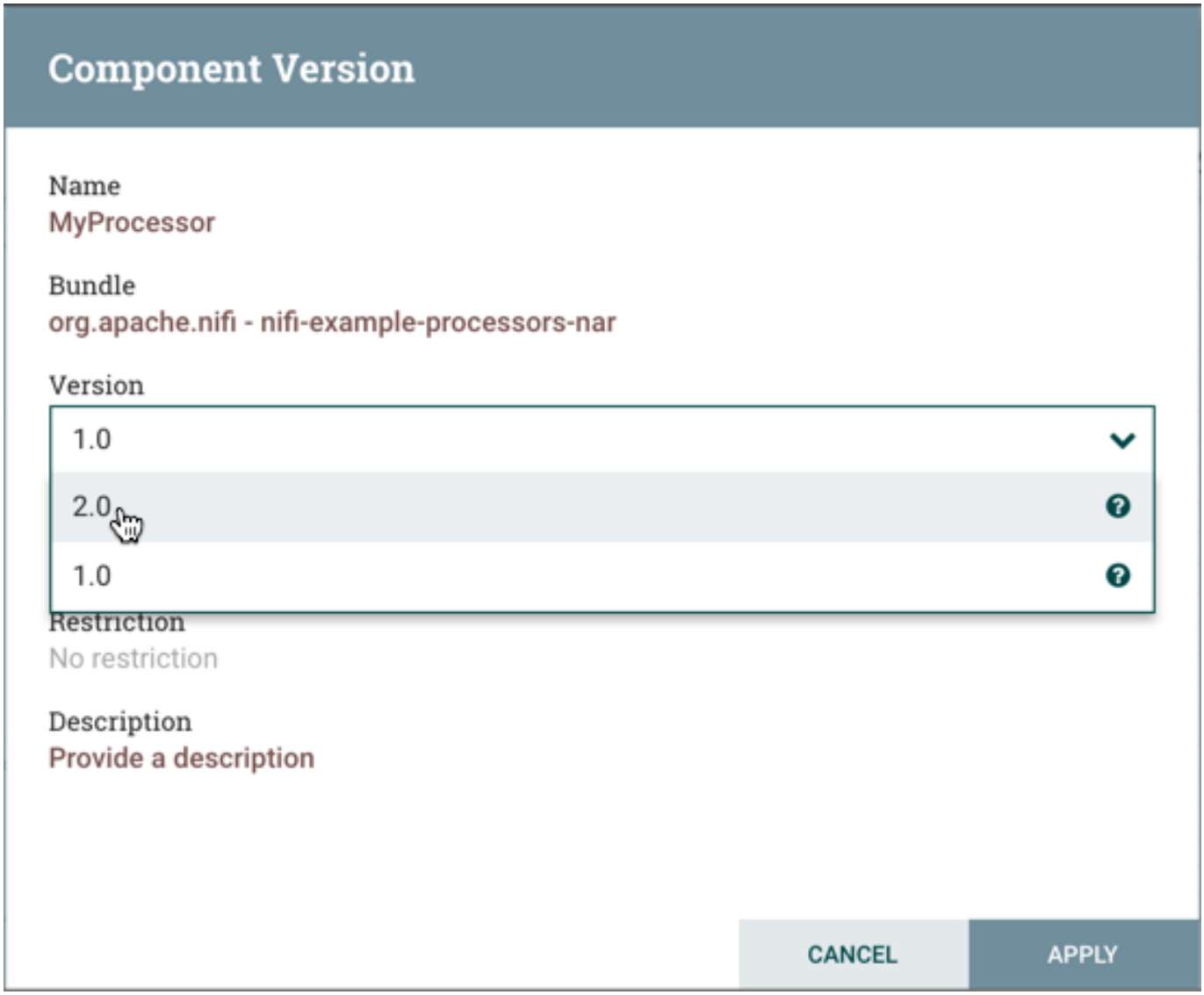

Рис.1.15.: Выбор версии компонента

При запущенном Процессоре в его контекстном меню опция “Configure” меняется на “View Configuration”, так как нельзя менять конфигурацию Процессора во время его работы. Для этого необходимо сначала остановить Процессор и дождаться завершения всех его активных задач.

Important: Ввод определенных символов не поддерживается и автоматически отфильтровывается при вводе. Следующие символы и любые непарные суррогатные коды Unicode не сохраняются ни в одной конфигурации:

```
[#x0], [#x1], [#x2], [#x3], [#x4], [#x5], [#x6], [#x7], [#x8], [#xB], [#xC], [#xE], [#xF], [#x10], [#x11], [
↪#x12], [#x13], [#x14], [#x15], [#x16], [#x17], [#x18], [#x19], [#x1A], [#x1B], [#x1C], [#x1D], [#x1E], [
↪#x1F], [#xFFFE], [#xFFFF]
```

1.6.1 Вкладка SETTINGS

При обращении к диалоговому окну настроек Процессора, оно открывается на первой вкладке “SETTINGS” (Рис.1.16.).

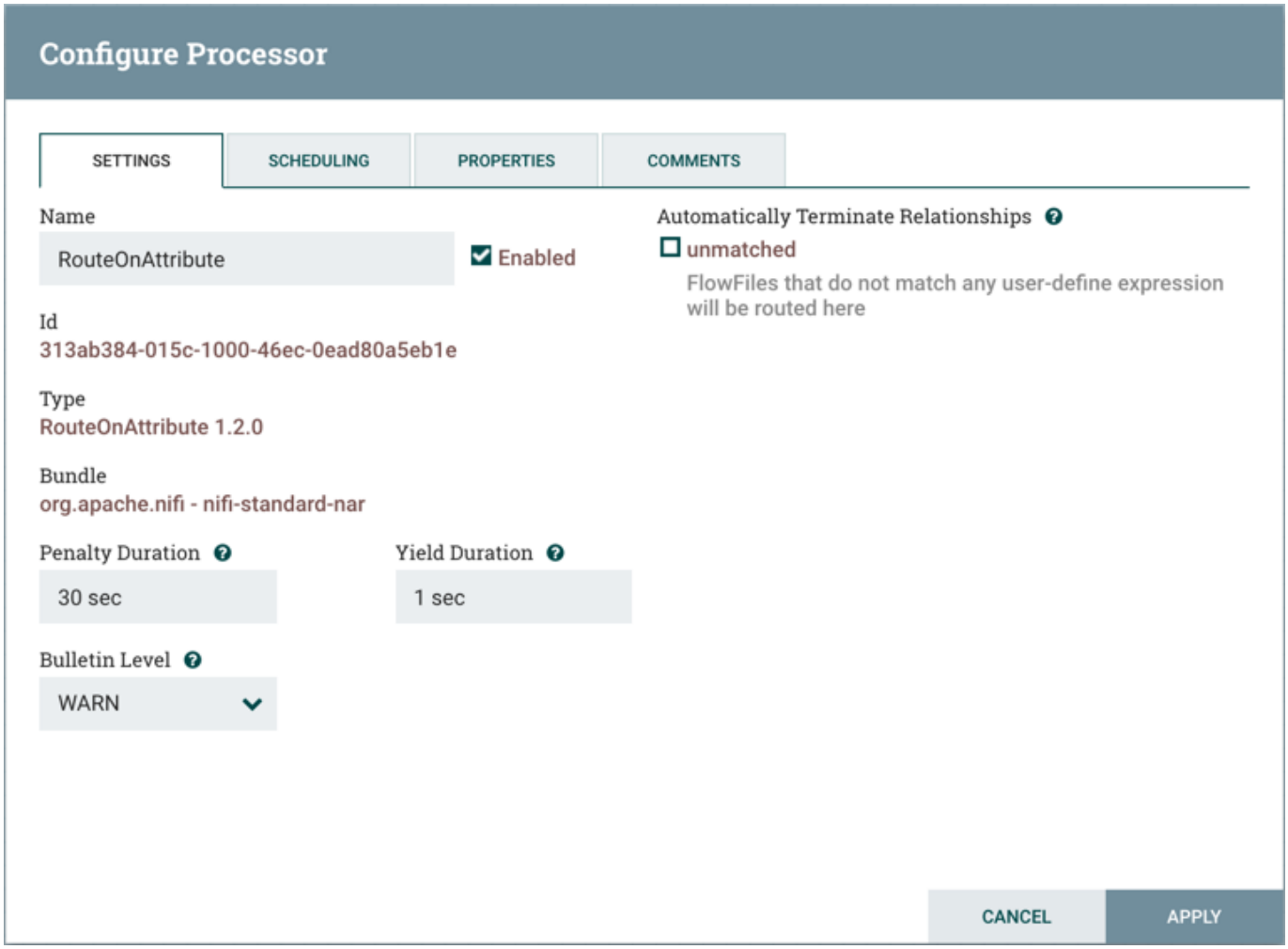

Рис.1.16.: Вкладка SETTINGS

Вкладка содержит несколько элементов конфигурации. В поле “Name” у DFM есть возможность изменить имя Процессора. По умолчанию оно совпадает с типом. Рядом с полем имени располагается флажок “Enabled”,

указывающий статус Процессора. При добавлении Процессора на рабочую область он активируется, и если его отключить, дальнейший запуск уже невозможен. Отключенное состояние используется для указания того, что, например, при запуске группы Процессоров данный (отключенный) Процессор следует исключить.

Ниже отображается уникальный идентификатор Процессора в поле "Id", его тип в поле "Type" и NAR в поле "Bundle". Данные значения не могут быть изменены.

Далее находятся поля "Penalty Duration" и "Yield Duration". Во время обычного процесса обработки потока данных (FlowFile) может произойти событие, указывающее, что данные могут быть обработаны не в данный момент, а позднее. Например, если Процессор должен передать данные удаленному сервису, но у сервиса уже есть одноименный файл. В таком случае Процессор может "оштрафовать" FlowFile сроком ожидания, что останавливает обработку данных потока на период времени, заданный в поле "Penalty Duration" (по умолчанию *30 секунд*).

Аналогичным образом Процессор может определить, что существует ситуация, при которой он больше не может выполнять действия независимо от обрабатываемых данных. Например, если Процессор должен отправить данные в удаленный сервис, но сервис не отвечает, Процессор при этом не может добиться какого-либо прогресса. В результате Процессор должен "уступить", а это отодвигает запланированный запуск Процессора на некоторый период времени, который задается в поле "Yield Duration" (значение по умолчанию *1 секунда*).

Последний настраиваемый параметр в левой части вкладки – "Bulletin Level". Всякий раз, когда Процессор записывает данные в свой журнал, он также создает бюллетень. В параметре указывается самый низкий уровень бюллетеня. По умолчанию установлено значение *WARN*, означающее, что в пользовательском интерфейсе отображаются все бюллетени с предупреждениями и ошибками.

В правой части вкладки "SETTINGS" находится "Automatically Terminate Relationships" с перечислением всех определенных Процессором связей и их описанием. Для того чтобы Процессор считался действительным и способным к запуску, каждая определенная им связь должна быть подключена к нисходящему компоненту, в противном случае она автоматически прерывается. При автоматическом прерывании связи любой направляемый в нее FlowFile удаляется из потока, и его обработка считается завершенной. Любая связь, которая уже подключена к нисходящему компоненту, не может быть автоматически завершена, для этого связь должна быть сначала удалена из использующих ее Соединений. Кроме того, у предназначенной к автоматическому прерыванию связи при ее добавлении к Соединению статус "auto-termination" отключается.

1.6.2 Вкладка SCHEDULING

"SCHEDULING" – вторая вкладка диалогового окна настроек Процессора (Рис.1.17.).

Вкладка содержит несколько элементов конфигурации:

- *Scheduling Strategy*
- *Concurrent Tasks*
- *Run Schedule*
- *Execution*
- *Run Duration*

Scheduling Strategy

В поле стратегии планирования "Scheduling Strategy" есть три возможных варианта планирования компонентов:

- *Timer driven* – режим по умолчанию – запуск Процессора осуществляется с регулярным интервалом, определенным опцией "Run Schedule";
- *Event driven* – в данном режиме Процессор инициируется с выполнением события, когда FlowFiles входят в Соединения, относящиеся к Процессору. Режим в настоящее время считается экспериментальным и не поддерживается всеми Процессорами. В данном режиме опция "Run Schedule" недоступна для

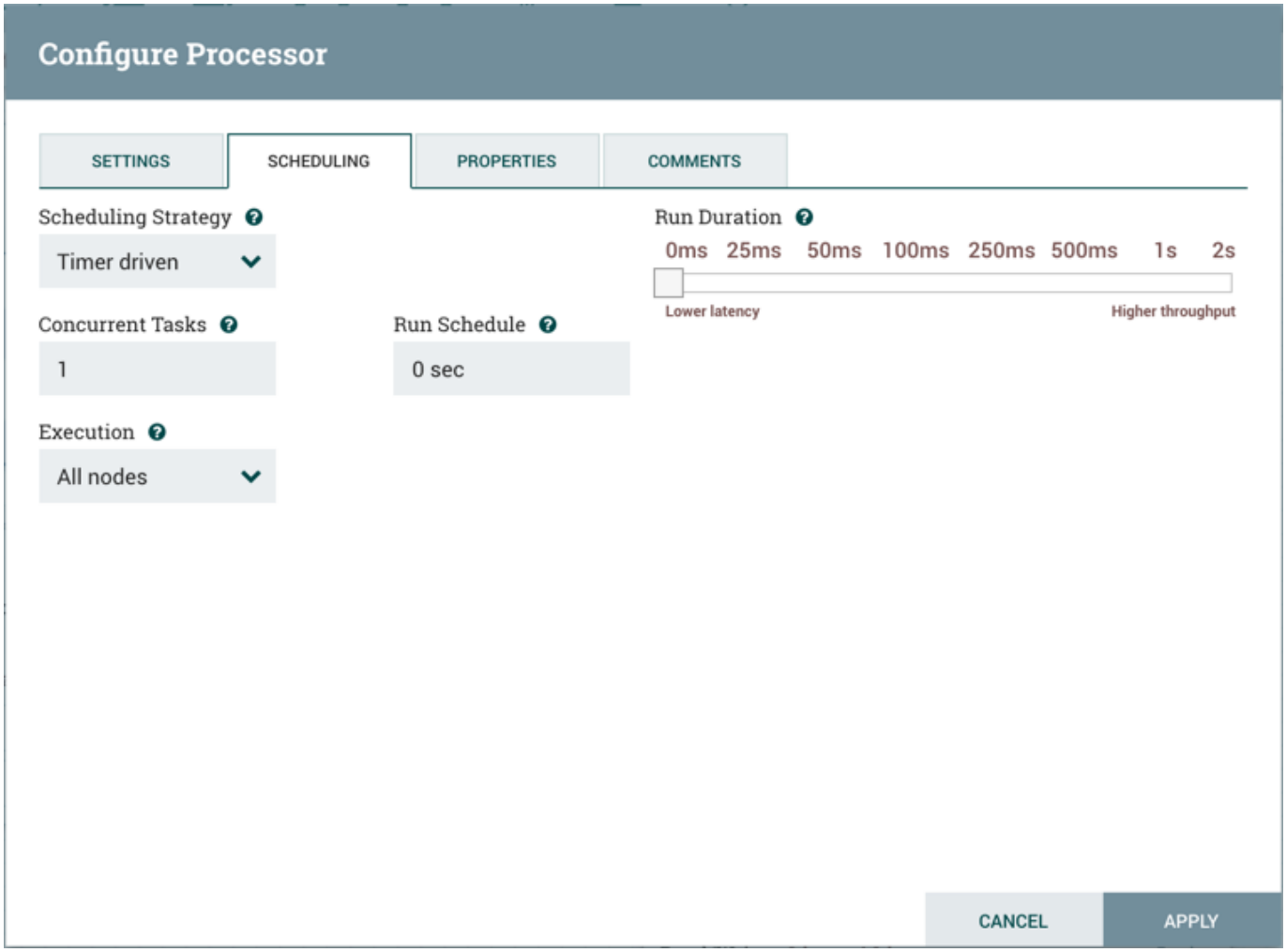

Рис.1.17.: Вкладка SCHEDULING

конфигурации, так как Процессор запускается не по периодам, а в результате выполнения события. Кроме того, это единственный режим, для которого параметр “Concurrent Tasks” может быть установлен равным *0*, так как количество потоков ограничено только размером Event-Driven Thread Pool, настроенным администратором;

- *CRON driven* – запуск Процессора осуществляется периодически, подобно режиму *Timer driven*. Однако *CRON driven* обеспечивает значительно большую гибкость за счет увеличения сложности конфигурации, представляющую собой строку из шести обязательных полей и одного опционального, разделенных пробелом.

Таблица1.2.: Конфигурация стратегии планирования CRON driven

Поле	Допустимые значения
Seconds	0-59
Minutes	0-59
Hours	0-23
Day of Month	1-31
Month	1-12 или JAN-DEC
Day of Week	1-7 или SUN-SAT
Year (опционально)	Пусто или 1970-2099

Значения задаются одним из следующих способов:

- *Number* – одно или несколько допустимых значений, разделенных запятыми;
- *Range* – диапазон значений в виде <number>-<number>;
- *Increment* – инкремент с использованием синтаксиса <start value>/<increment>. Например, в поле “Minutes” значение *0/15* обозначает последовательность минут *0*, *15*, *30* и *45*.

Кроме того, могут быть использованы специальные символы:

- Символ * – означает, что все допустимые значения действительны;
- Символ ? – означает, что может быть задано не характерное значение (допустипо в полях “Day of Month” и “Day of Week”);
- Символ L – можно добавить *L* к одному из значений дня недели, чтобы указать последнее вхождение этого дня в месяце. Например, *1L* обозначает последнее воскресенье месяца.

Примеры:

- Строка `0 0 13 * * ?` указывает, что необходимо запланировать запуск Процессора в 13:00 каждый день;
- Строка `0 20 14 ? * MON-FRI` указывает, что необходимо запланировать запуск Процессора в 14:20 с понедельника по пятницу;
- Строка `0 15 10 ? * 6L 2011-2017` указывает, что необходимо запланировать запуск Процессора в 10:15 в последнюю пятницу каждого месяца в период с 2011 по 2017 год.

Дополнительную информацию с примерами можно найти в документации Quartz по ссылке Chron Trigger Tutorial.

Concurrent Tasks

Параметр конфигурации “Concurrent Tasks” – параллельные задачи – определяет количество потоков, используемых Процессором, то есть количество одновременно обрабатываемых FlowFiles. Увеличение значения, как правило, позволяет Процессору обрабатывать больше данных за тот же промежуток времени. Однако это достигается засчет использования системных ресурсов, которые в таком случае не могут использоваться другими

Процессорами. Параметр по существу контролирует, сколько ресурсов системы должно быть выделено для конкретного Процессора.

Поле “Concurrent Tasks” доступно для большинства Процессоров. Однако существуют некоторые типы Процессоров, которые можно запланировать только с одной параллельной задачей.

Run Schedule

Параметр “Run Schedule” определяет расписание запуска Процессора. Допустимые значения для поля зависят от выбранной стратегии планирования “Scheduling Strategy”. При стратегии *Event driven* поле “Run Schedule” недоступно. При стратегии *Timer driven* значение представляет собой единицу времени. Например, *1 second* или *5 mins*. Значение по умолчанию *0 sec* означает, что Процессор должен работать как можно чаще при наличии данных для обработки. Условие верно для любой продолжительности времени со значением *0* (независимо от единицы времени, то есть *0 sec*, *0 mins*, *0 days*). Объяснение значений, применимых к стратегии *CRON driven*, приведено в описании самой стратегии.

Execution

Параметр “Execution” используется для определения узла, на котором запланирован запуск Процессора. Выбор значения *All Nodes* приводит к планированию запуска Процессора на каждом узле кластера. Значение *Primary Node* приводит к тому, что запуск Процессора планируется только на первичном узле. Настроенные на *Primary Node* Процессоры помечаются значком с буквой “P” рядом с пиктограммой самого Процессора (Рис.1.18.).

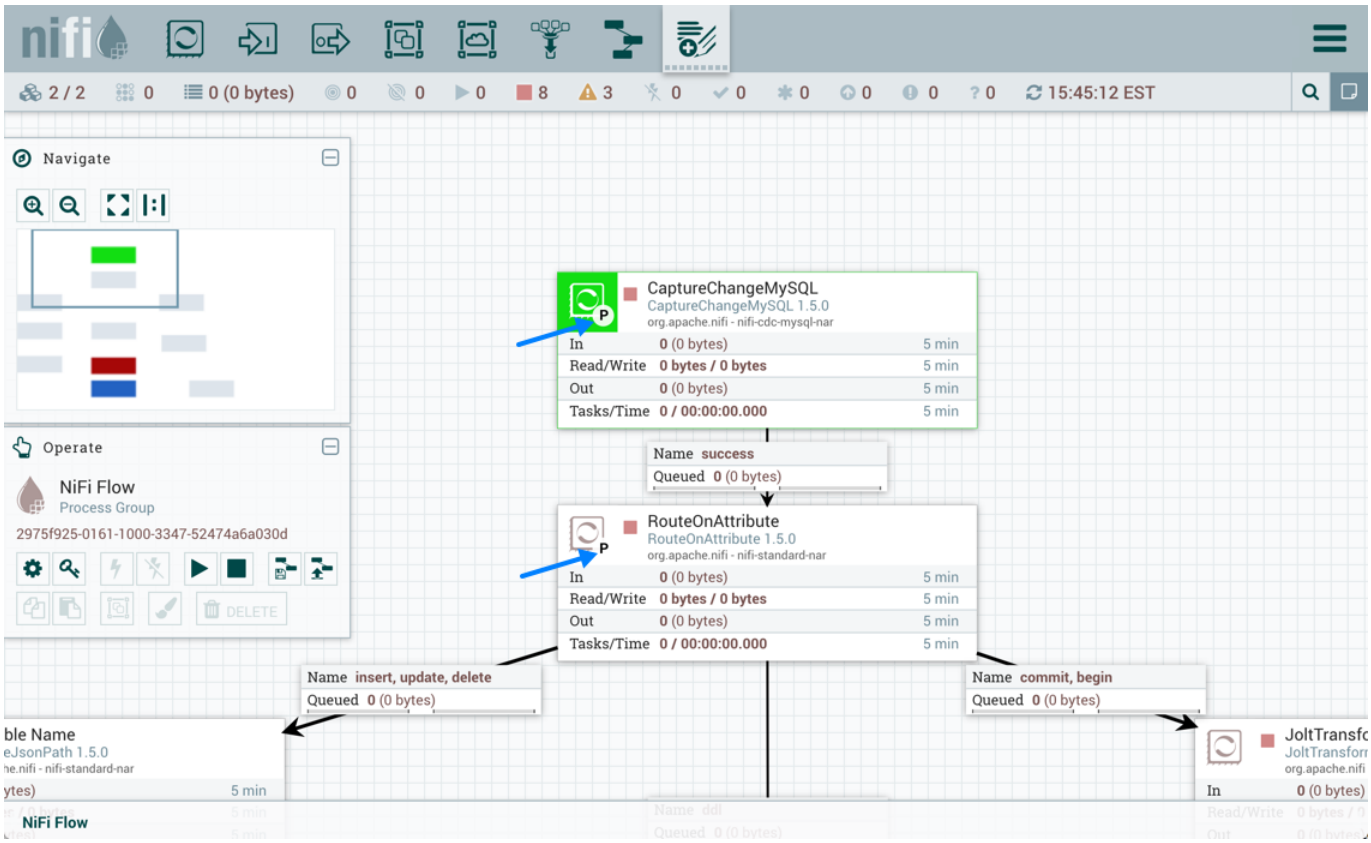

Рис.1.18.: Процессоры, настроенные на Primary Node

Для быстрого определения настроенных на первичный узел Процессоров значок “P” также отображается на вкладке “Processors” на странице “Summary” (Рис.1.19.).

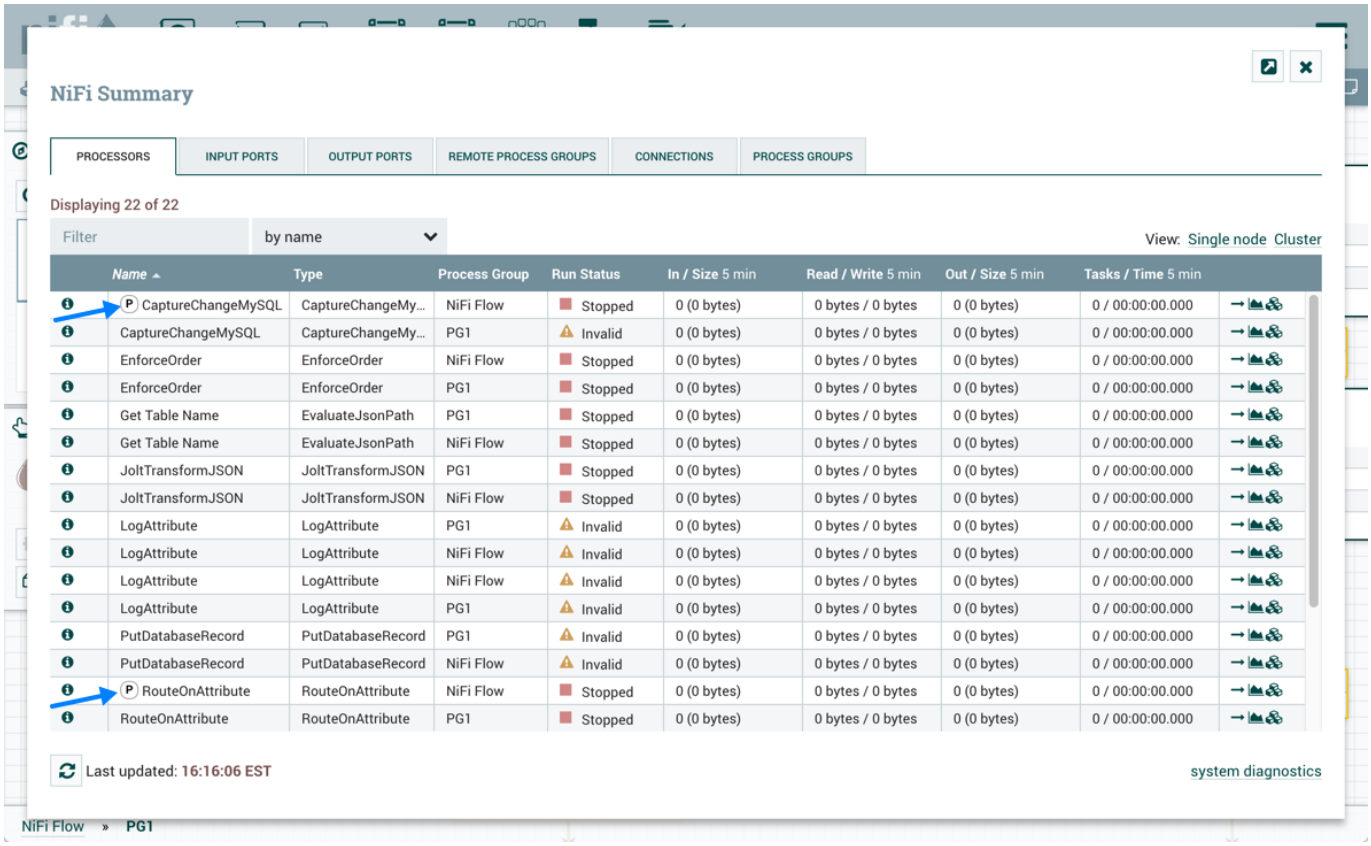

Рис.1.19.: Процессоры, настроенные на Primary Node

Run Duration

Правая часть вкладки “SCHEDULING” содержит ползунок для управления параметром “Run Duration”, определяющий длительность повторных запусков Процессора. Левая часть ползунка помечена как нижняя грань латентности “Lower latency”, правая – как наиболее высокая пропускная способность “Higher throughput”.

При завершении работы Процессор обновляет репозиторий, чтобы передать FlowFiles следующему Соединению. Обновление хранилища является ресурсоемким процессом, поэтому, чем больше работы может быть выполнено перед обновлением репозитория, тем больше работы Процессор может обработать (более высокая пропускная способность). Однако, это означает, что следующий Процессор не может начать обработку задействованных потоков, пока предыдущий процесс не обновит репозиторий, что приводит к увеличению латентности (время, необходимое для обработки FlowFile от начала до конца, увеличивается). В результате, ползунок параметра “Run Duration” предоставляет спектр, из которого DFM выбирает более низкую задержку или более высокую пропускную способность.

1.6.3 Вкладка PROPERTIES

Вкладка свойств “PROPERTIES” предоставляет механизм для настройки поведения Процессора. Свойства не задаются по умолчанию и каждому типу Процессора определяются характерные ему оцпии, имеющие смысл для конкретного использования. Далее приведены свойства для Процессора *RouteOnAttribute* (Рис.1.20.).

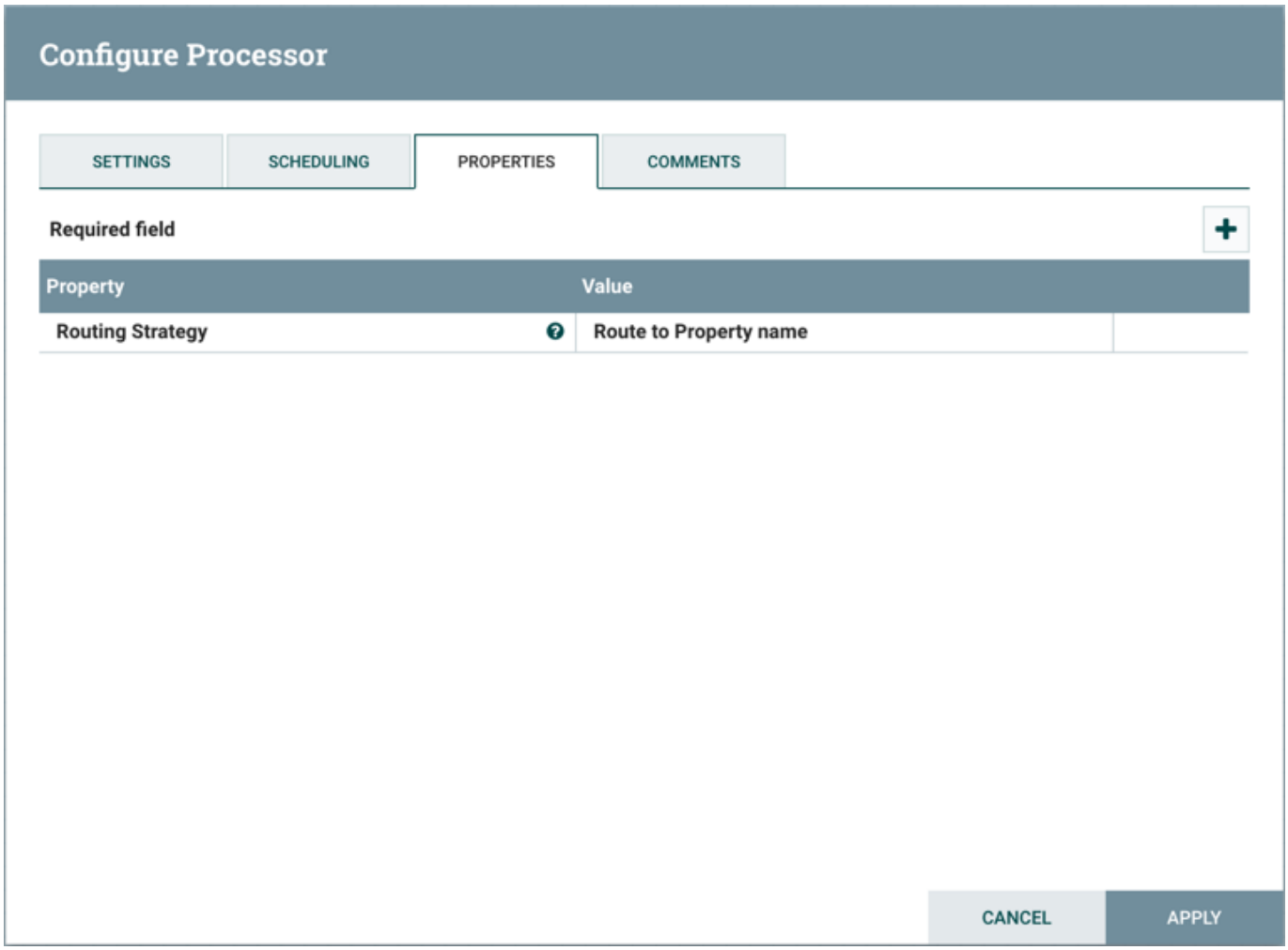

Рис.1.20.: Вкладка PROPERTIES

Данный Процессор имеет одно свойство по умолчанию – “Routing Strategy”, заданное на “Route to Property name”. Рядом с наименованием свойства находится символ знака вопроса, и как и в других местах интерфейса символ обозначает информационную справку для пользователя, при наведении курсора на которую в данном случае выдается дополнительная информация о свойстве и установленном значении по умолчанию, а также его исторические значения.

Клик по значению свойства позволяет DFM изменить его. В зависимости от допустимых значений для конкретного свойства предоставляется либо раскрывающийся список, либо открывается текстовое поле для ввода (Рис.1.21.).

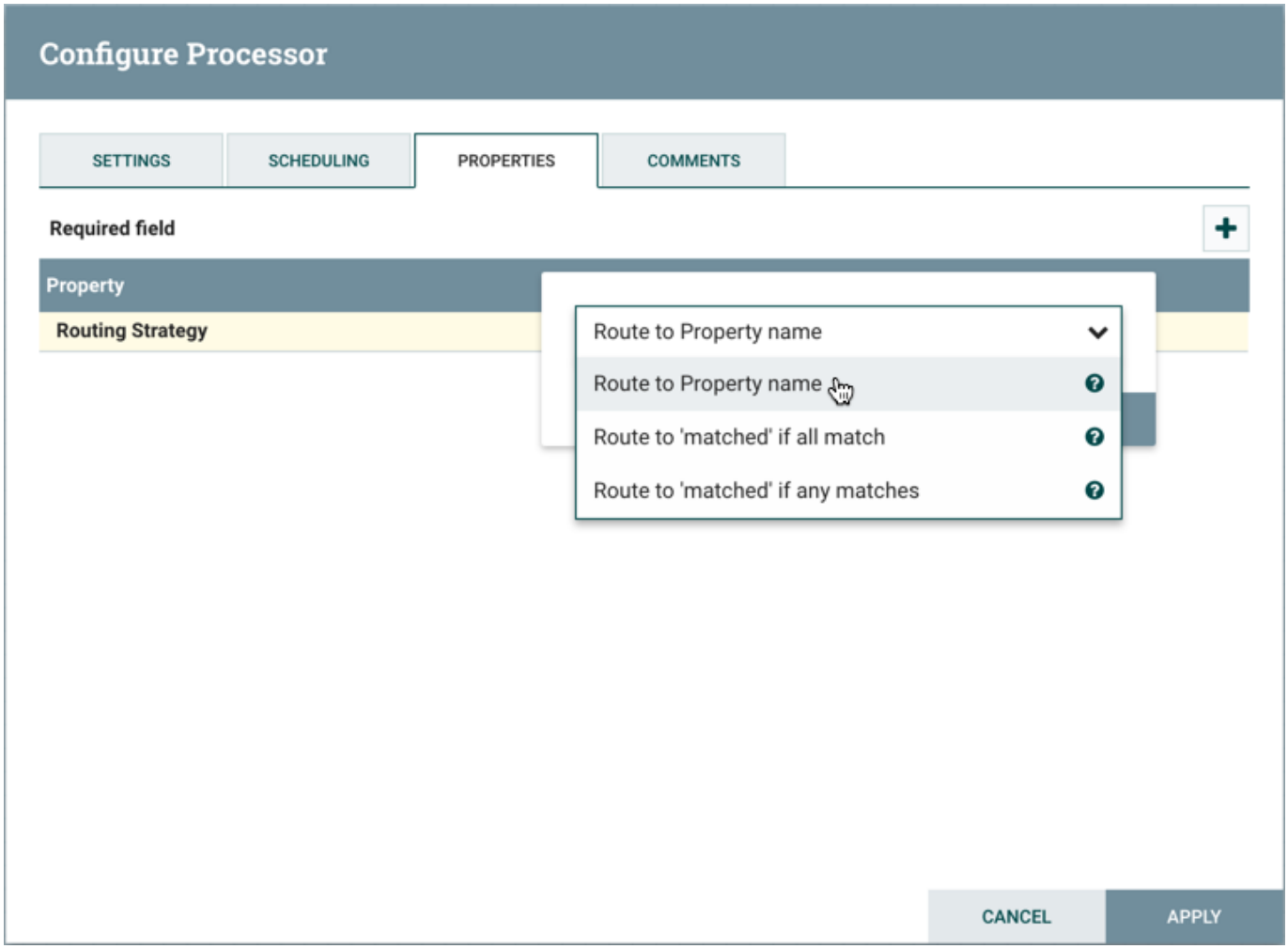

Рис.1.21.: Выбор значения свойства

В правом верхнем углу вкладки расположена кнопка добавления свойства “New Property”, при нажатии на которую открывается диалоговое окно для ввода имени и значения нового свойства. Не всеми Процессорами допускаются пользовательские свойства User-Defined, и в случае их назначения Процессор становится недействителен (Рис.1.22.).

После добавления свойства User-Defined в правой части его строки появляется значок удаления, при нажатии на который свойство удаляется из Процессора.

Некоторые Процессоры, например, *UpdateAttribute*, имеют встроенный пользовательский интерфейс. Для перехода к нему необходимо нажать кнопку “Advanced”, которая появляется в нижней части окна настройки у подобных Процессоров.

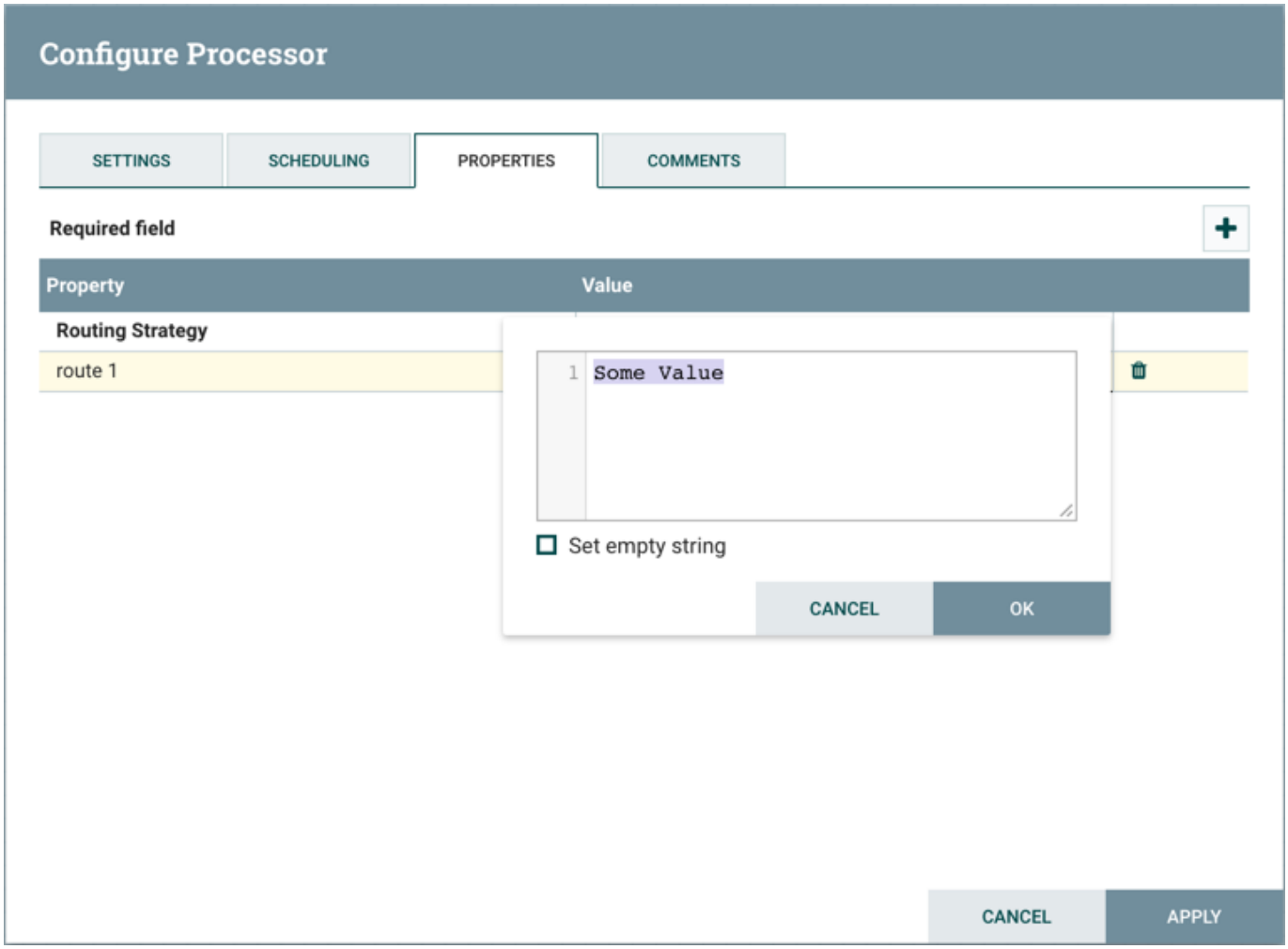

Рис.1.22.: Добавление свойства

Так же некоторые Процессоры имеют свойства, ссылающиеся на другие компоненты, например, на Controller Services, которые также требуют настройки. Например, Процессор *GetHTTP* имеет свойство *SSLContextService*, которое ссылается на контроллер *StandardSSLContextService*. В случае, когда DFM необходимо настроить свойство, но при этом еще не создан и не настроен контроллер, у DFM есть возможность сделать это сразу на месте, как показано далее на рисунке (Рис.1.23.).

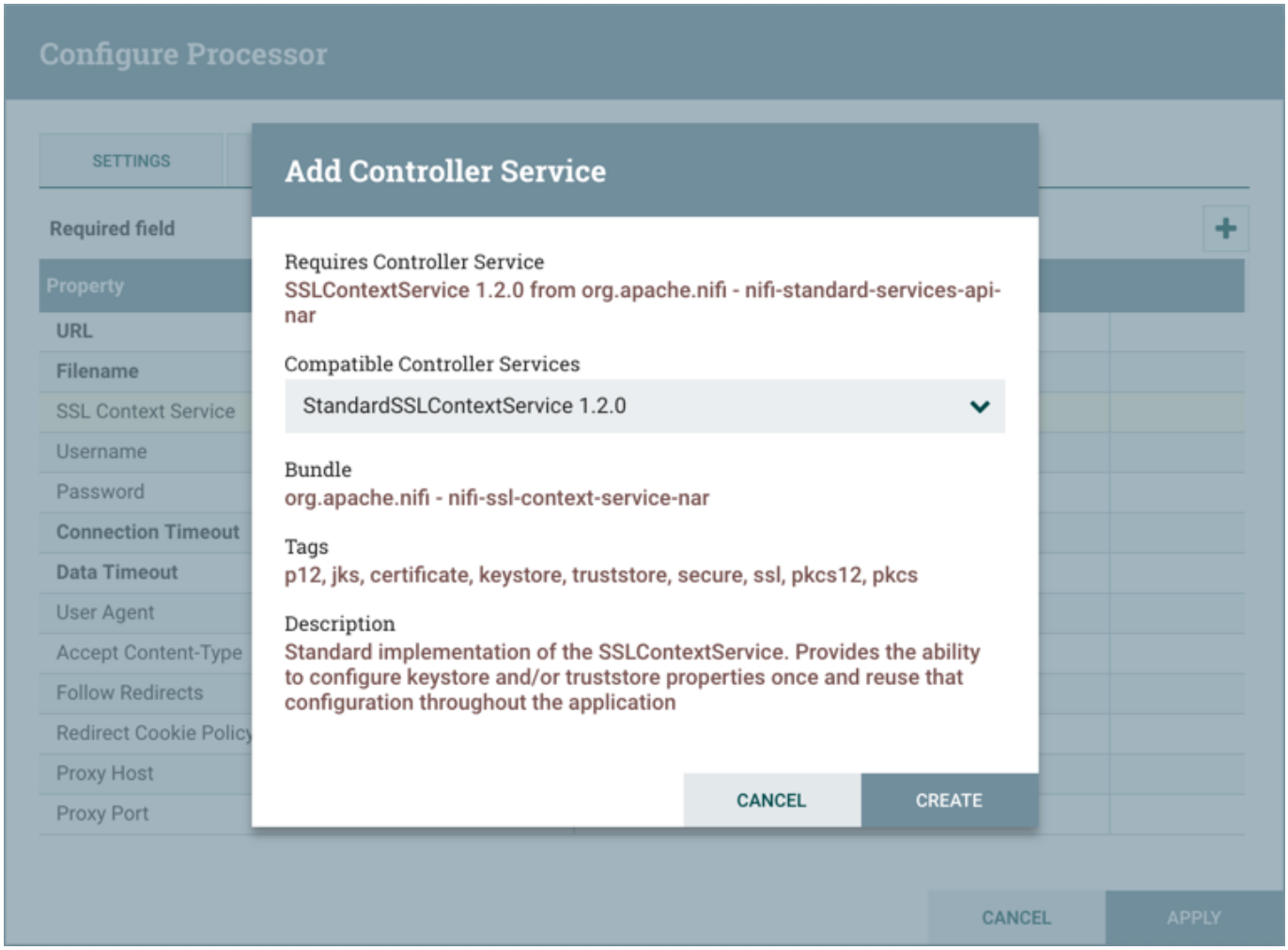

Рис.1.23.: Настройка контроллера через свойство Процессора

1.6.4 Вкладка COMMENTS

Последней вкладкой диалогового окна конфигурации Процессора является вкладка “COMMENTS”, предоставляющая пользователям область для добавления комментариев к компоненту. Использование вкладки необязательно (Рис.1.24.).

1.7 Пользовательские свойства с использованием Expression Language

Expression Language (язык выражений) **NiFi** можно использовать для отображения значений атрибутов FlowFile, сравнения их с другими значениями и управления ими при создании и настройке потоков данных. Сведения о языке выражений по ссылке Expression Language Guide.

В дополнение к использованию атрибутов FlowFile, системных свойств и свойств среды с помощью Express

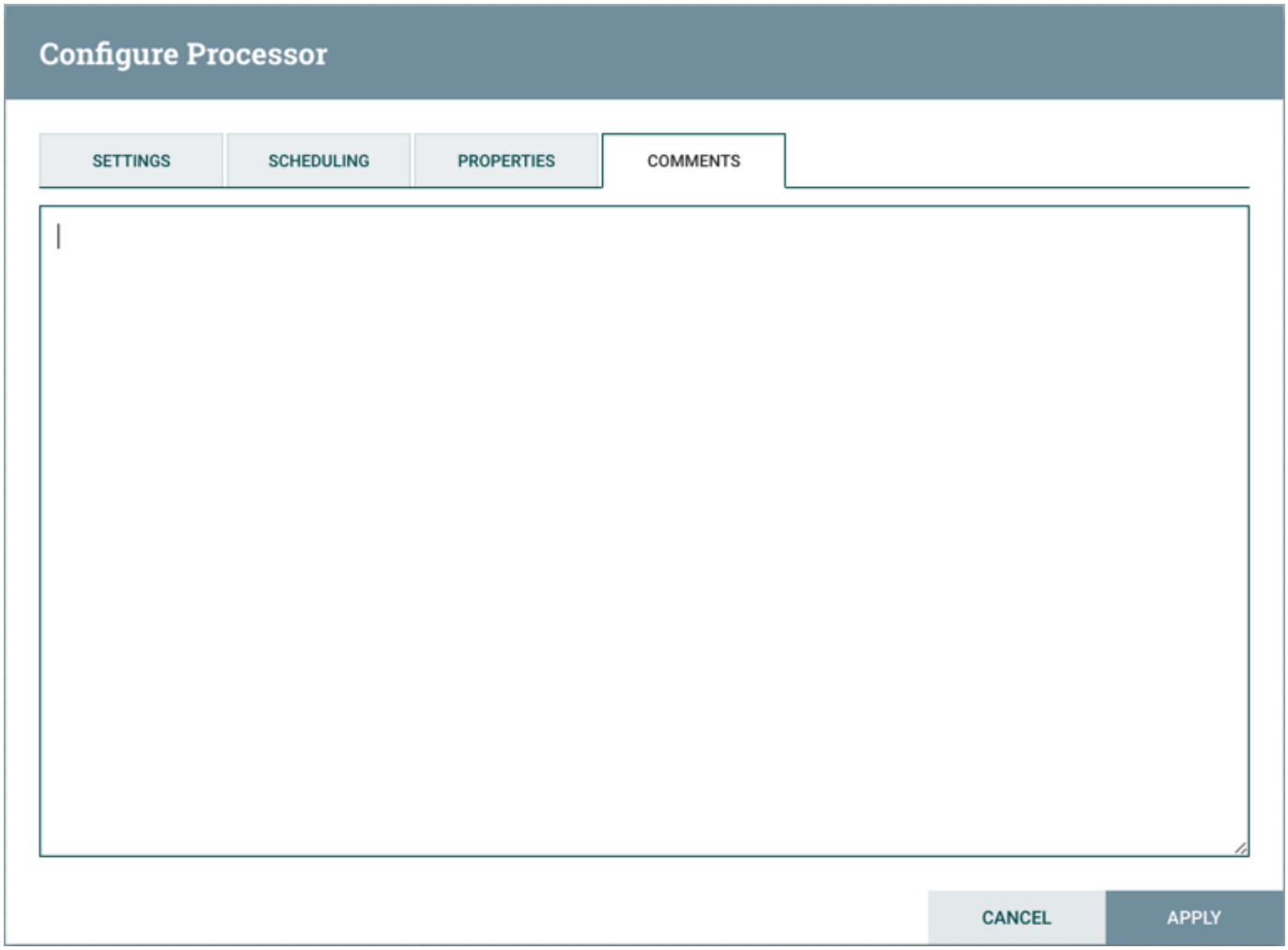

Рис.1.24.: Вкладка COMMENTS

Language также можно определить пользовательские свойства, что обеспечивает большую гибкость в управлении и обработке потоков данных. Также возможно создание пользовательских свойств для соединения, сервера и сервиса с целью упрощения настройки потока данных.

Свойства **NiFi** имеют приоритет разрешения, о котором следует знать при создании пользовательских свойств:

- Атрибуты процессора;
- Свойства FlowFile;
- Атрибуты FlowFile;
- Из реестра переменных:
  - Пользовательские свойства;
  - Системные свойства;
  - Переменные среды операционной системы.

Important: При создании пользовательских свойств необходимо убедиться, что каждое такое свойство содержит отдельное значение с целью предотвращения его переопределения существующими свойствами среды, системными свойствами или атрибутами FlowFile

Есть два способа использования и управления пользовательскими свойствами:

- В интерфейсе NiFi через окно переменных “Variables”;
- По ссылке *nifi.properties*.

1.7.1 Окно переменных Variables

Переменные могут быть созданы и настроены в пользовательском интерфейсе **NiFi** в любом поле, поддерживающем Expression Language. **NiFi** автоматически подбирает новые или измененные переменные, созданные в UI.

Для перехода к диалоговому окну “Variables” необходимо кликнуть правой кнопкой мыши по пустому пространству рабочей области приложения и выбрать пункт контекстного меню “Variables” (Рис.1.25.).

При выборе группы процессов переход к “Variables” также доступен из контекстного меню (Рис.1.26.).

Диалоговое окно создания и настроек пользовательских свойств “Variables” представлено на рисунке (Рис.1.27.).

Создание переменной

Для создания новой переменной необходимо в окне “Variables” нажать кнопку с символом “+” и в открывшейся форме ввести наименование (Рис.1.28.).

Затем нажать кнопку “OK” и в новом окне ввести значение переменной (Рис.1.29.).

После чего в экранной форме “Variables” нажать кнопку “Apply”, результатом действия которой является иформационное окно по обновлению/созданию переменной (Рис.1.30.).

При этом запускается процесс обновления переменной (идентификация затронутых компонентов, остановка затронутых процессоров и т.д.). Например, в разделе “Referencing Processors” теперь отображается процессор “PutFile-Root”. Выбор имени процессора в списке приводит к переходу к указанному процессору на рабочей области. А при просмотре свойств процессора свойство *Directory* ссылается на созданную переменную *${putfile_ dir}* (Рис.1.31.).

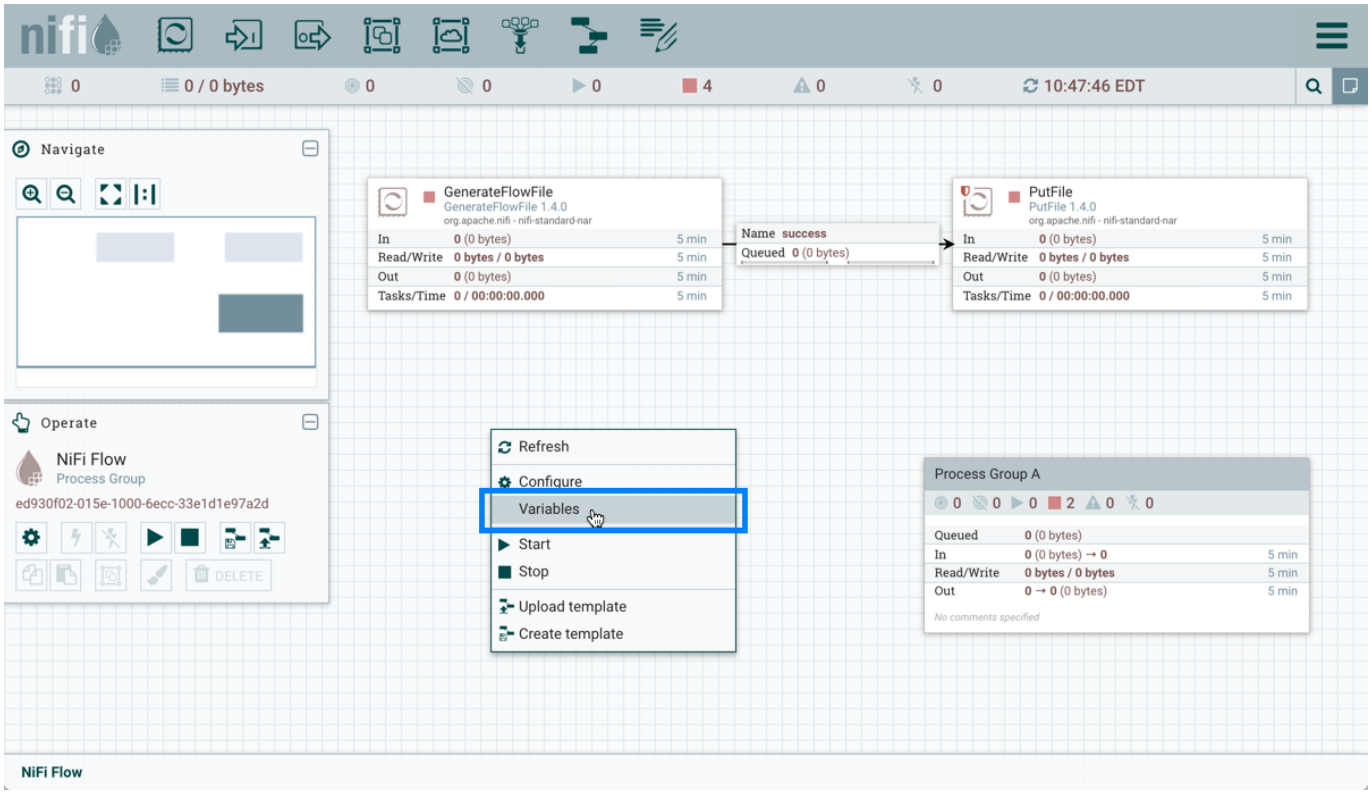

Рис.1.25.: Переход к “Variables” с рабочей области NiFi

Область действия переменной

Область действия переменных определяется группой процессов, в которой они заданы, и доступны любому Процессору, определенному на данном уровне и ниже (то есть любому наследованному Процессору).

При этом переменные в наследованной группе переопределяют значения в родительской группе. В частности, если переменная задана в группе *root*, а также получает иное значение внутри группы процессов, то в таком случае компоненты внутри группы процессов используют значение, определенное непосредственно в самой группе процессов.

Например, переменная *putfile_dir* существует в группе процессов *root*, и в то же время создается другая переменная *putfile_dir* в группе процессов *A*. В таком случае, если один из компонентов в группе процессов *A* ссылается на переменную *putfile_dir*, то указываются обе переменные, но *putfile_dir* из группы *root* при этом перечеркнута, так как она переопределена (Рис.1.32.).

Значение переменной может быть изменено только в группе процессов, в которой она создана (данная группа указывается в верхней части окна “Variables”). Для изменения переменной, определенной в другой группе процессов, необходимо выбрать значок стрелки в строке интересующей переменной (Рис.1.33.).

При этом происходит переход к экранной форме “Variables” группы процессов, создавшей переменную (Рис.1.34.).

Разрешения переменных

Разрешения переменных основаны исключительно на привилегиях, настроенных для соответствующей группы процессов. Например, если у пользователя нет доступа к просмотру группы процессов (“View a process group”), окно “Variables” для данной группы процессов не может быть открыто (Рис.1.35.).

При наличии у пользователя прав доступа к просмотру группы процессов, но при этом отсутствии доступа

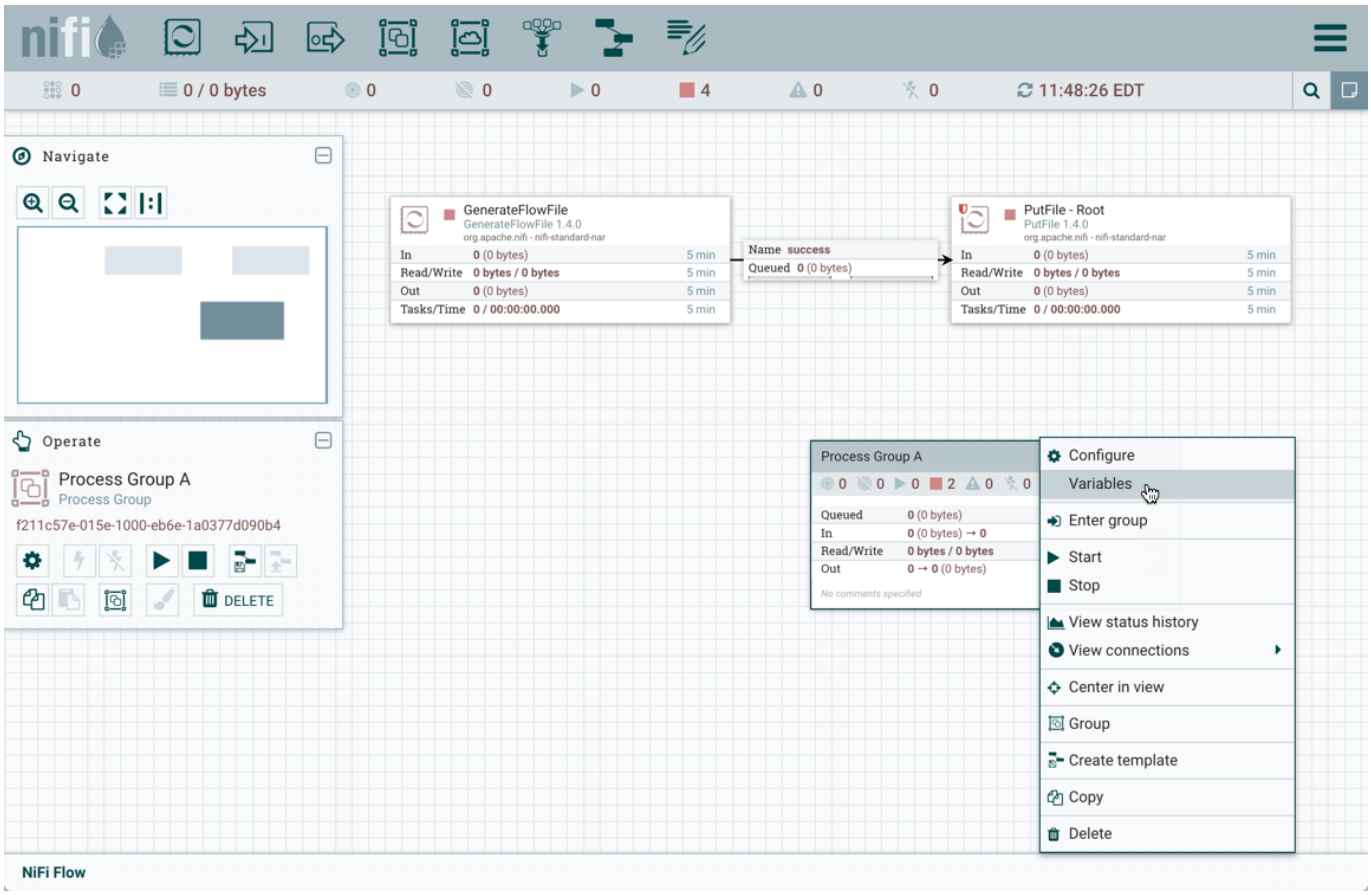

Рис.1.26.: Переход к “Variables” через группу процессов

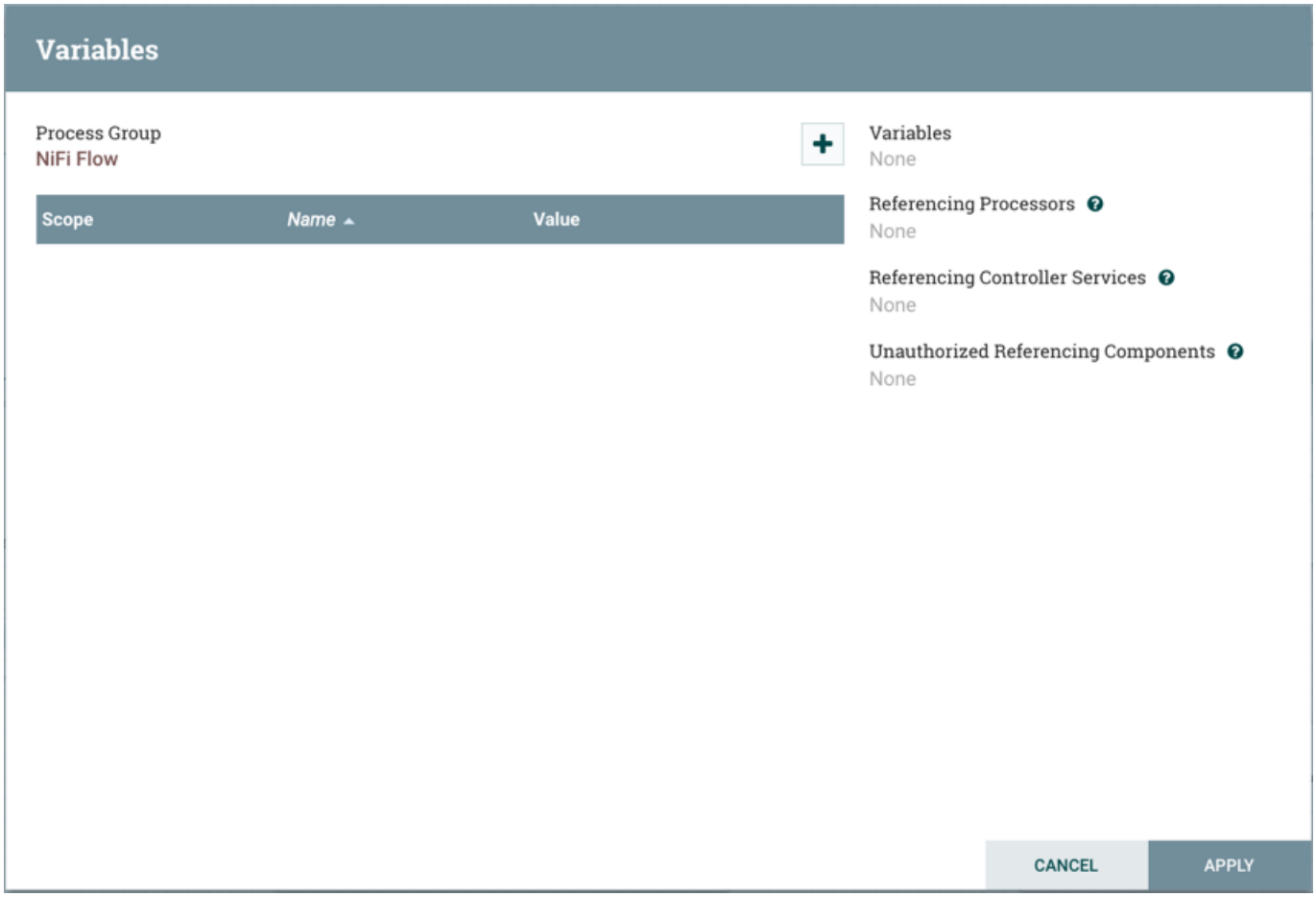

Рис.1.27.: Диалоговое окно “Variables”

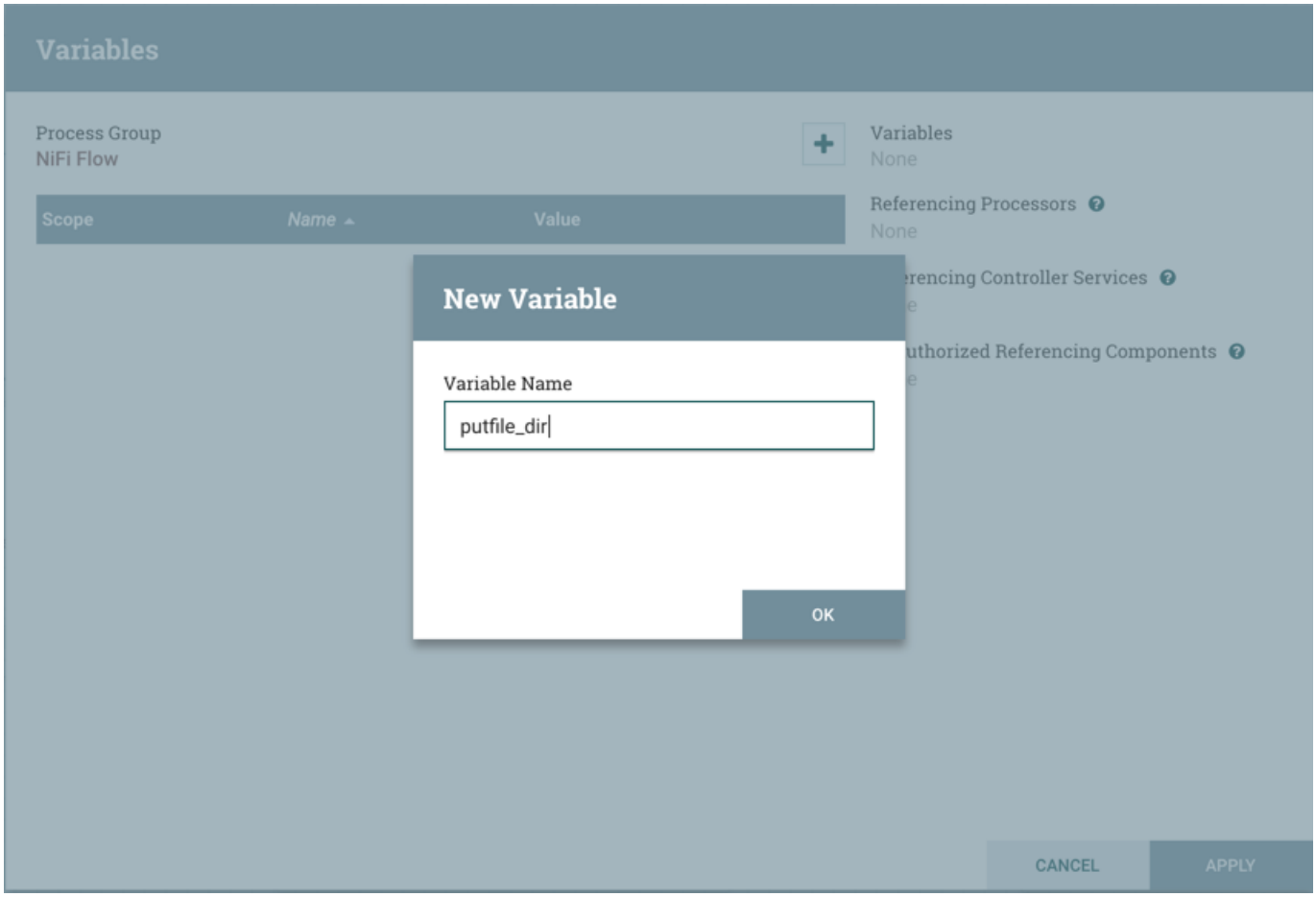

Рис.1.28.: Наименование новой переменной

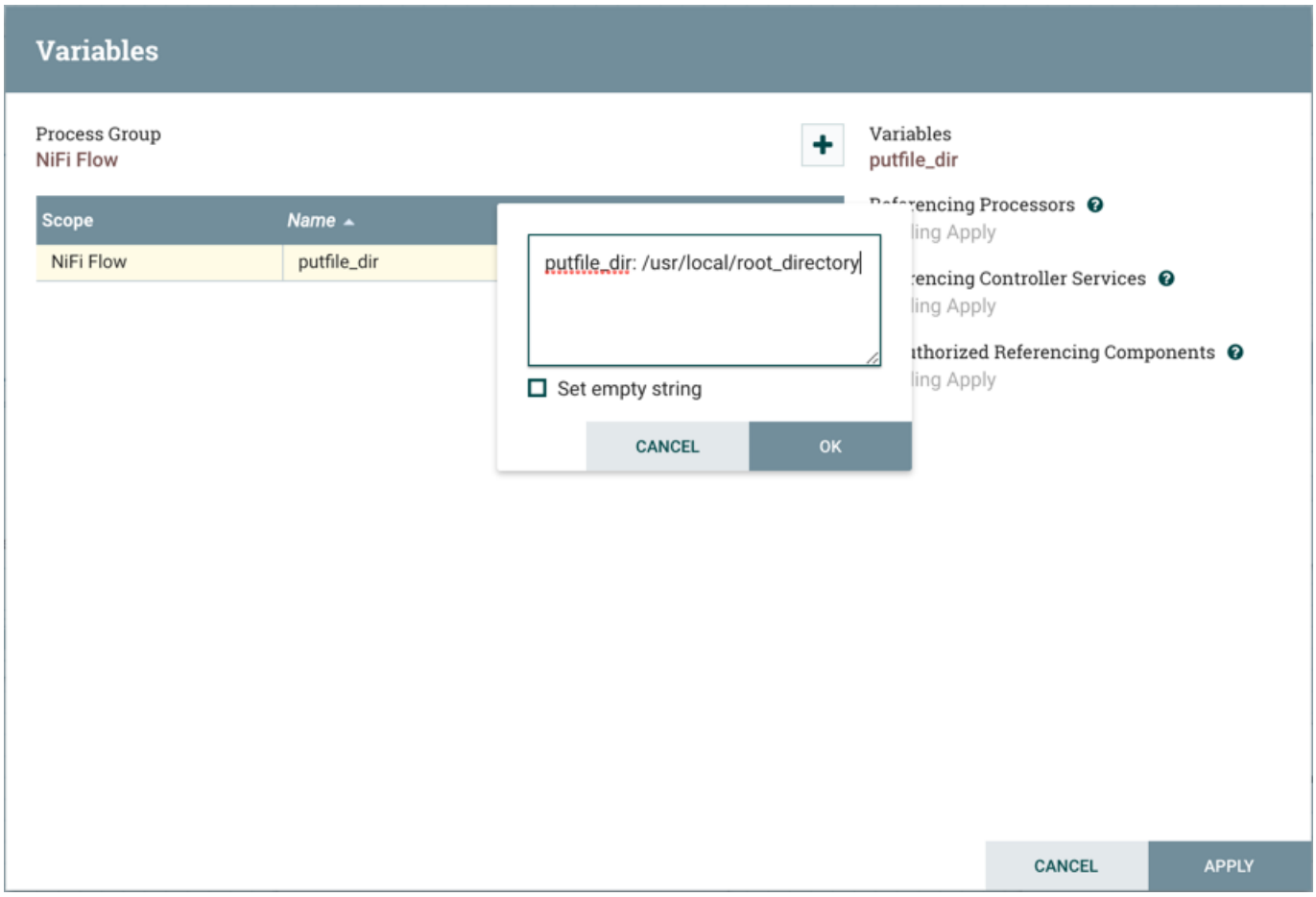

Рис.1.29.: Значение новой переменной

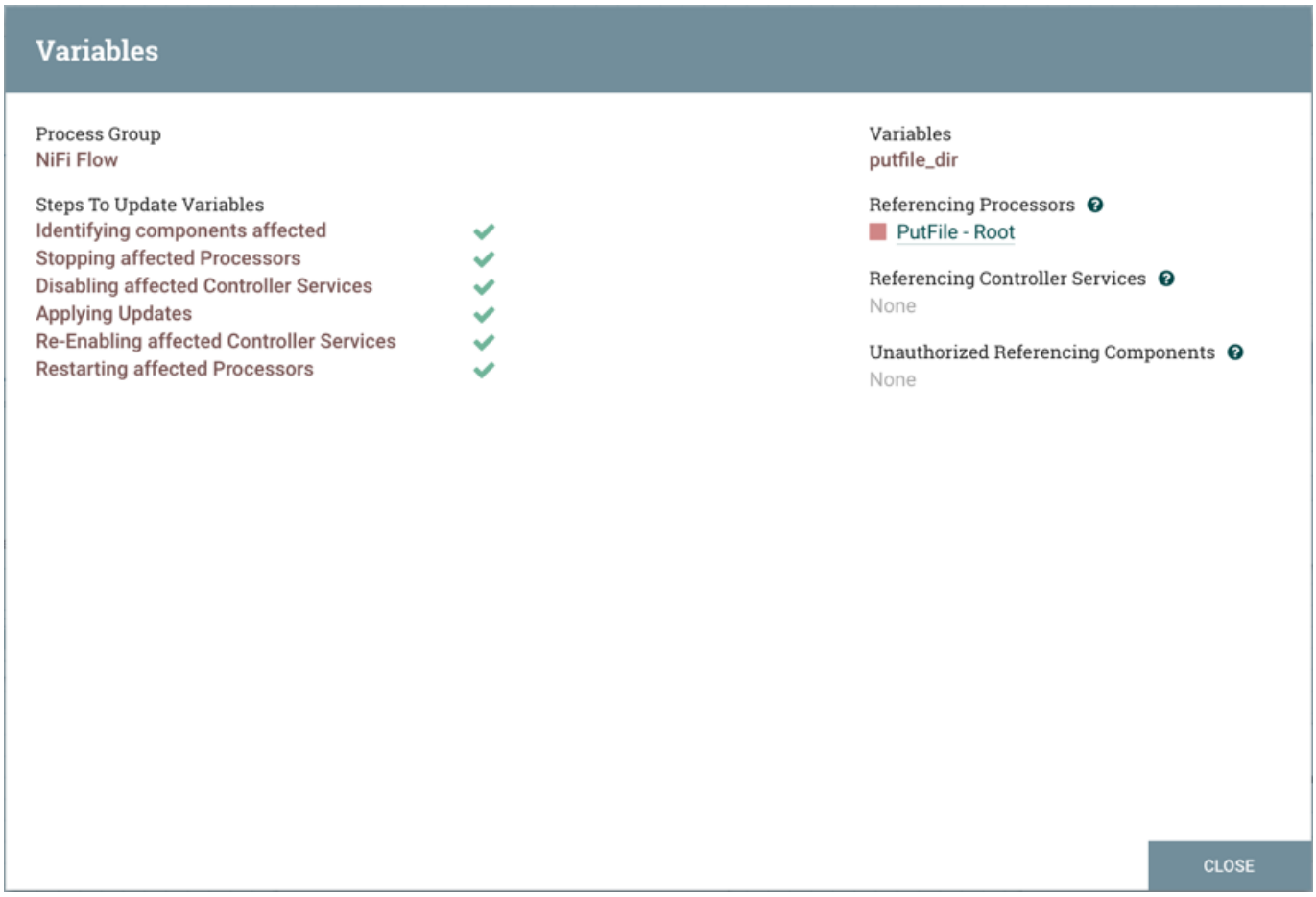

Рис.1.30.: Обновление переменной

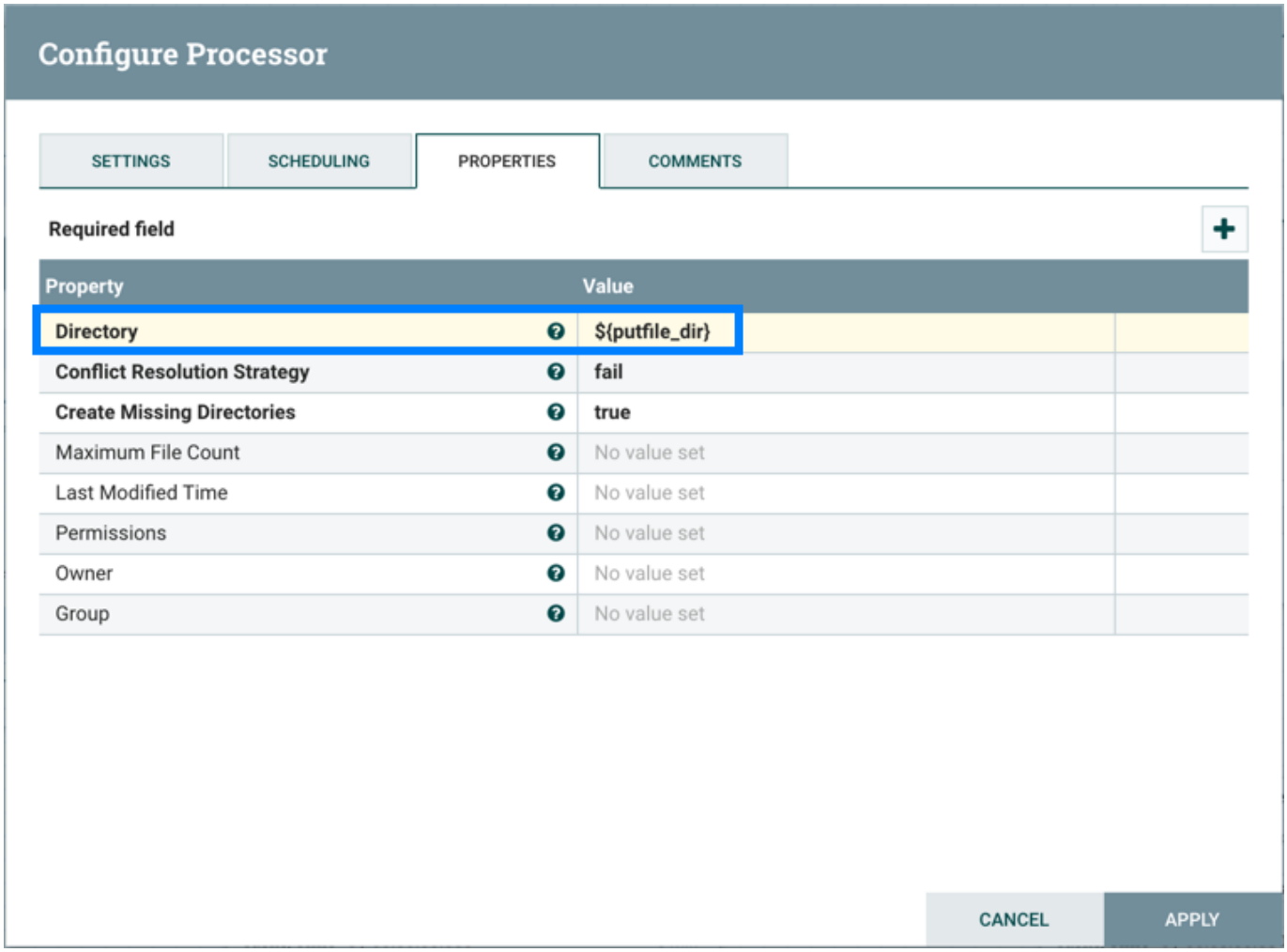

Рис.1.31.: Переменная в свойствах процессора

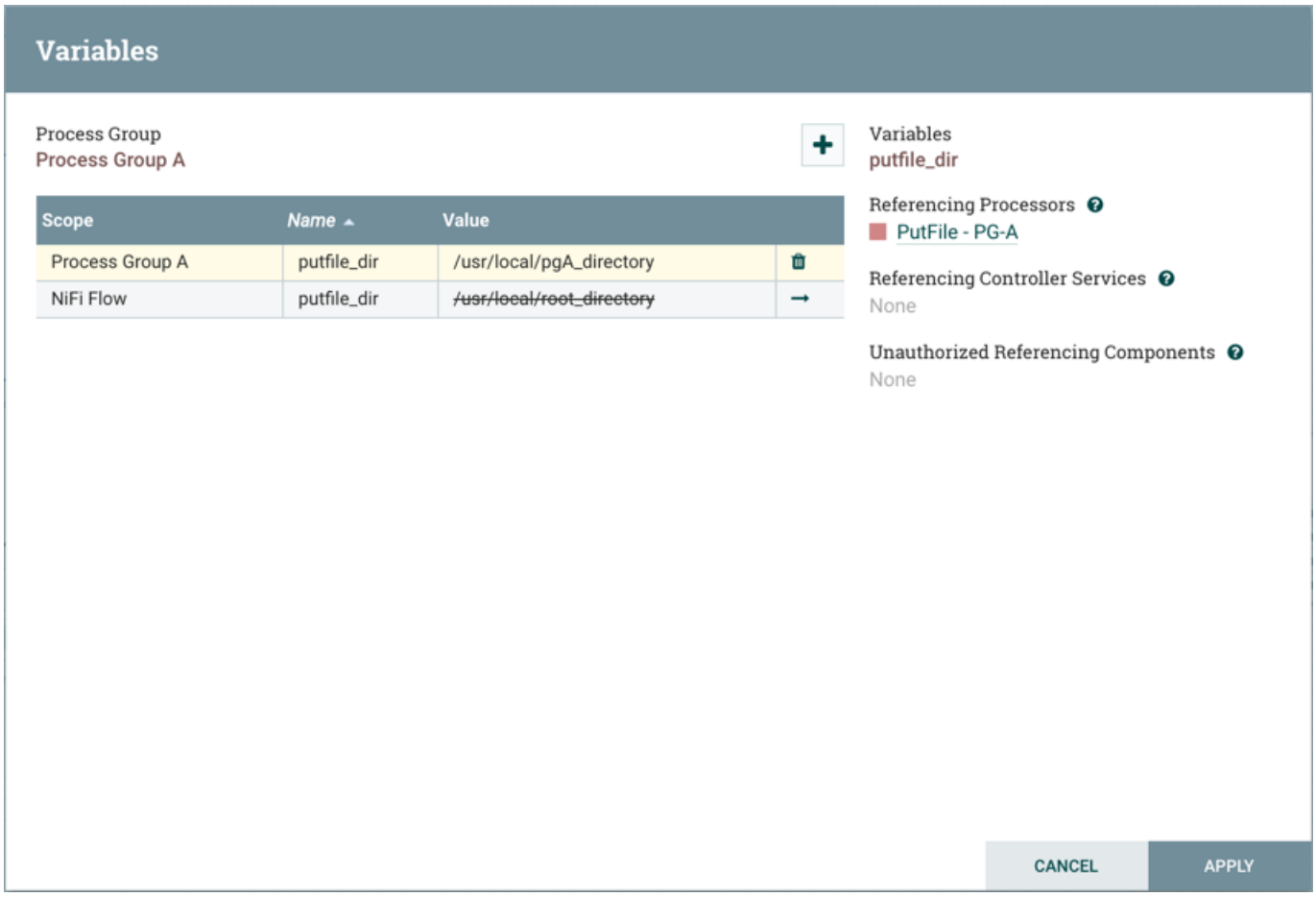

Рис.1.32.: Переопределение переменной

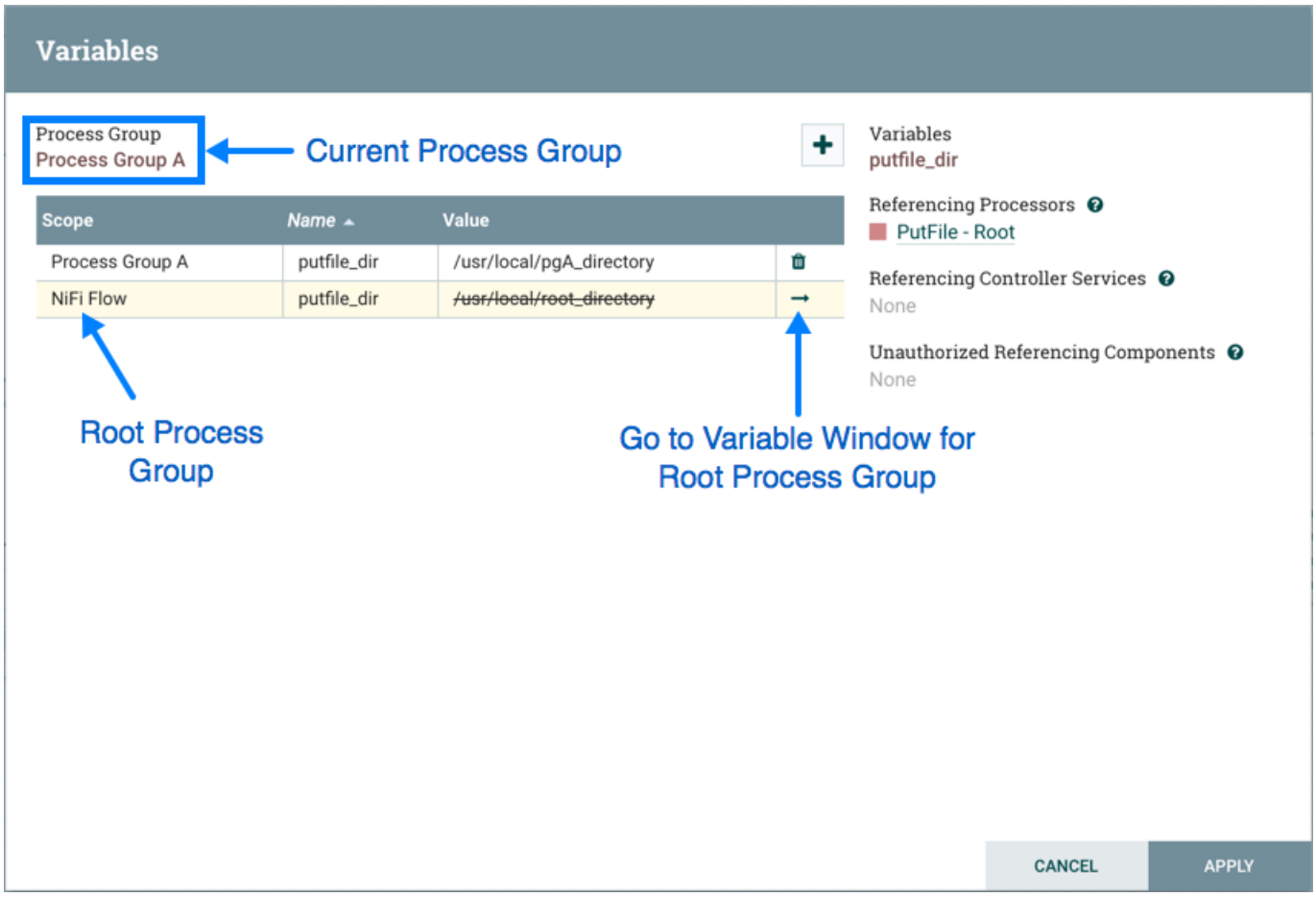

Рис.1.33.: Изменение переменной

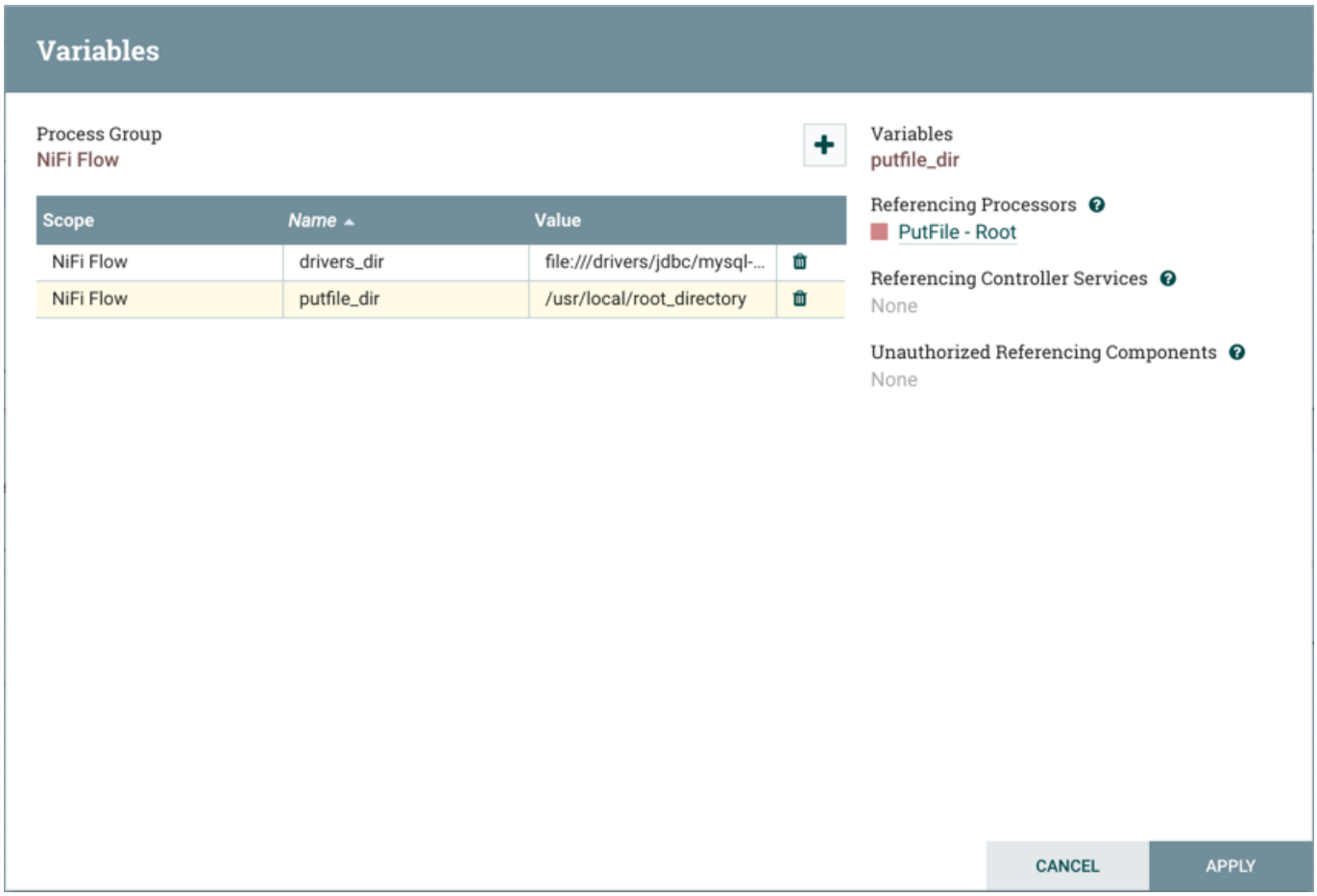

Рис.1.34.: Переход к группе процессов, создавшей переменную

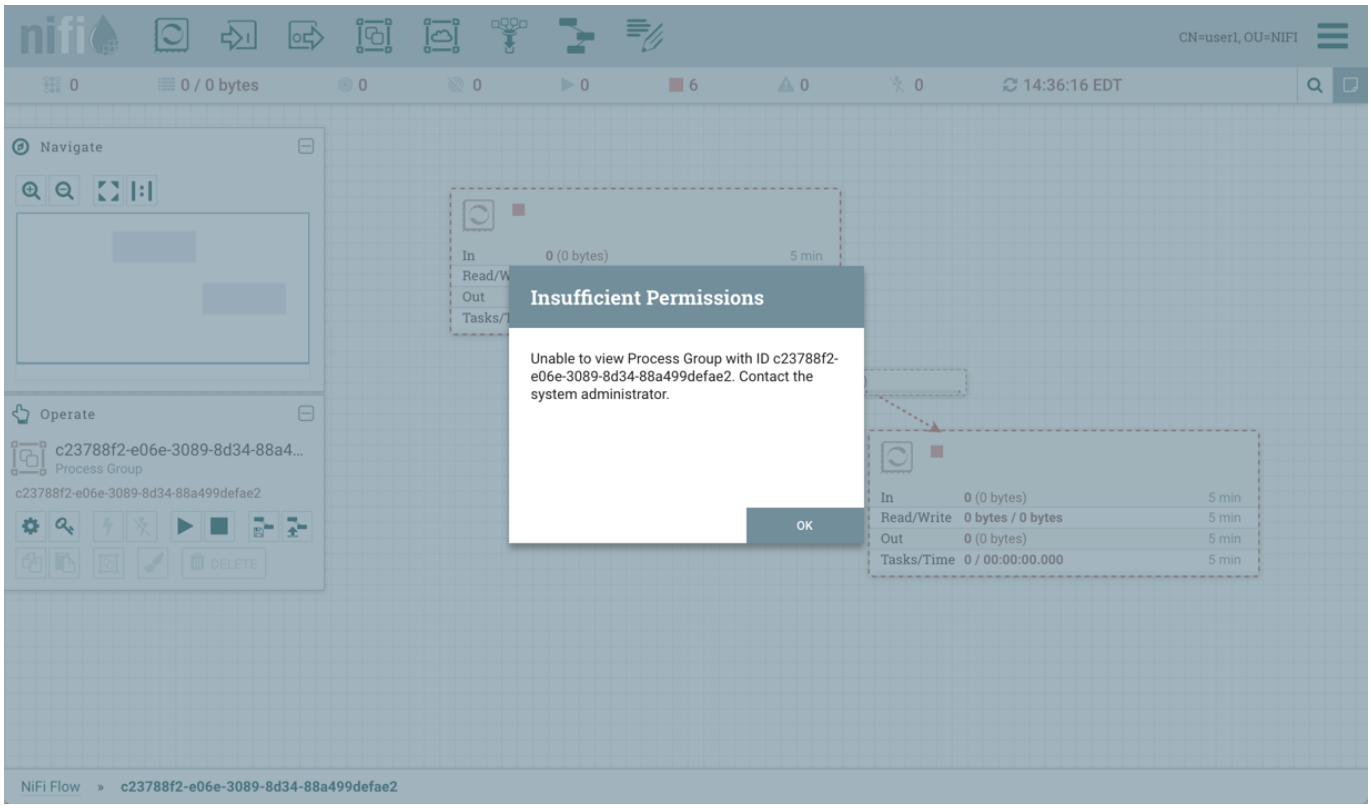

Рис.1.35.: Отсутствие доступа к экранной форме “Variables”

к изменению настроек (“Modify the process group”), переменные так же можно только просматривать, но не изменять.

Сведения об управлении привилегиями компонентов приведены в разделе Политики доступа документа Руководство администратора по работе с сервисом Nifi.

Controller Services

В экранной форме “Variables” также отображаются ссылки на контроллеры (Рис.1.36.).

При выборе контроллера происходит переход к экранной форме сервиса окна конфигурации (Рис.1.37.).

Ссылки на компоненты

В случаях, когда компоненту, ссылающемуся на переменную, не предоставлены права на просмотр или изменение, в окне “Variables” отображается UUID данного компонента (Рис.1.38.).

В приведенном примере свойство *property1* ссылается на процессор, в котором у пользователя *user1* нет прав доступа на просмотр (Рис.1.39.).

1.7.2 Ссылка nifi.properties

Пользовательские свойства могут быть созданы и настроены с использованием ссылки *nifi.properties*. Для этого необходимо определить один или несколько наборов пар ключ/значение и передать их системному администратору. После добавления новых пользовательских свойств важно убедиться, что поле *nifi.variable.registry.properties* в файле *nifi.properties* обновлено.

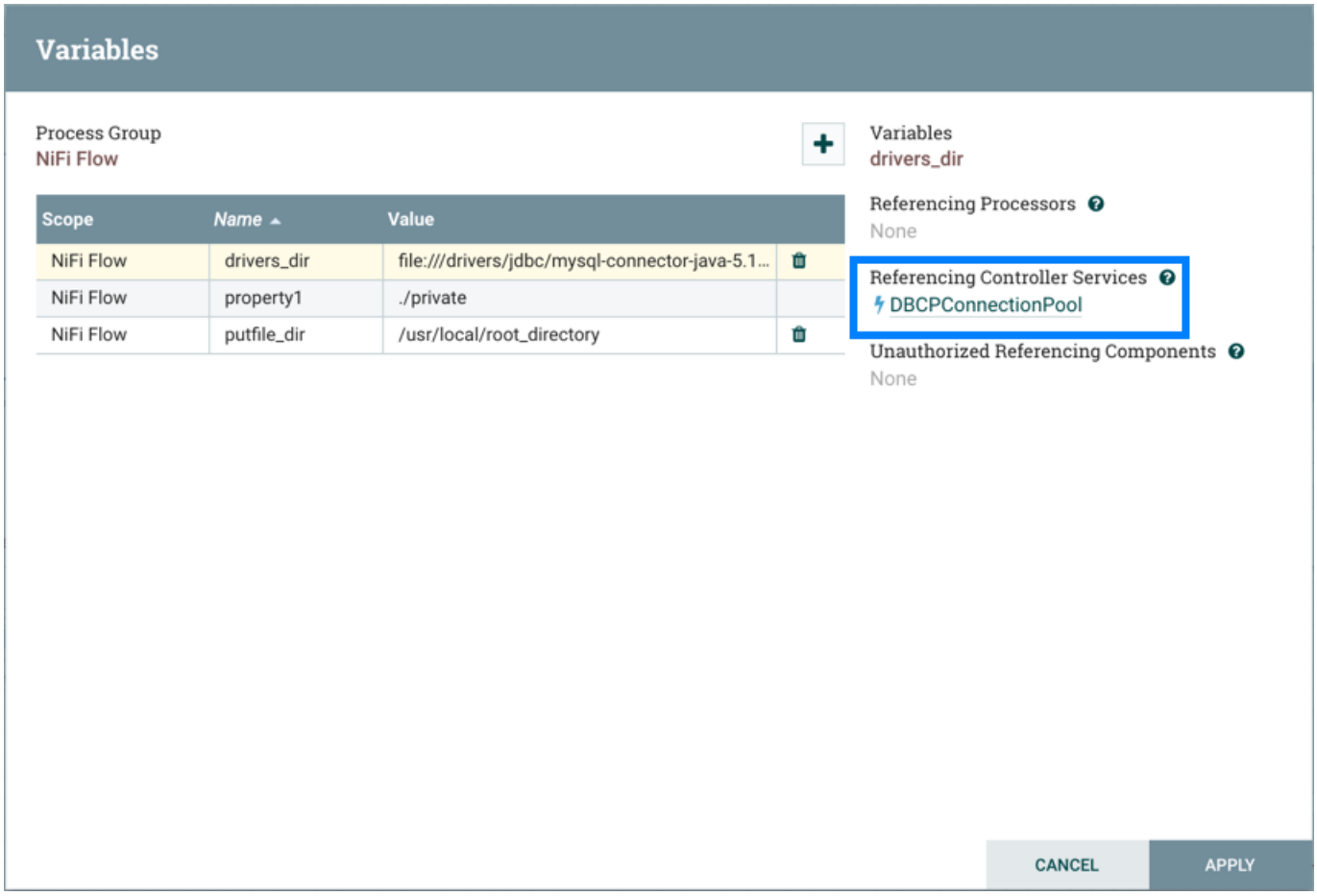

Рис.1.36.: Ссылка на Controller Services в “Variables”

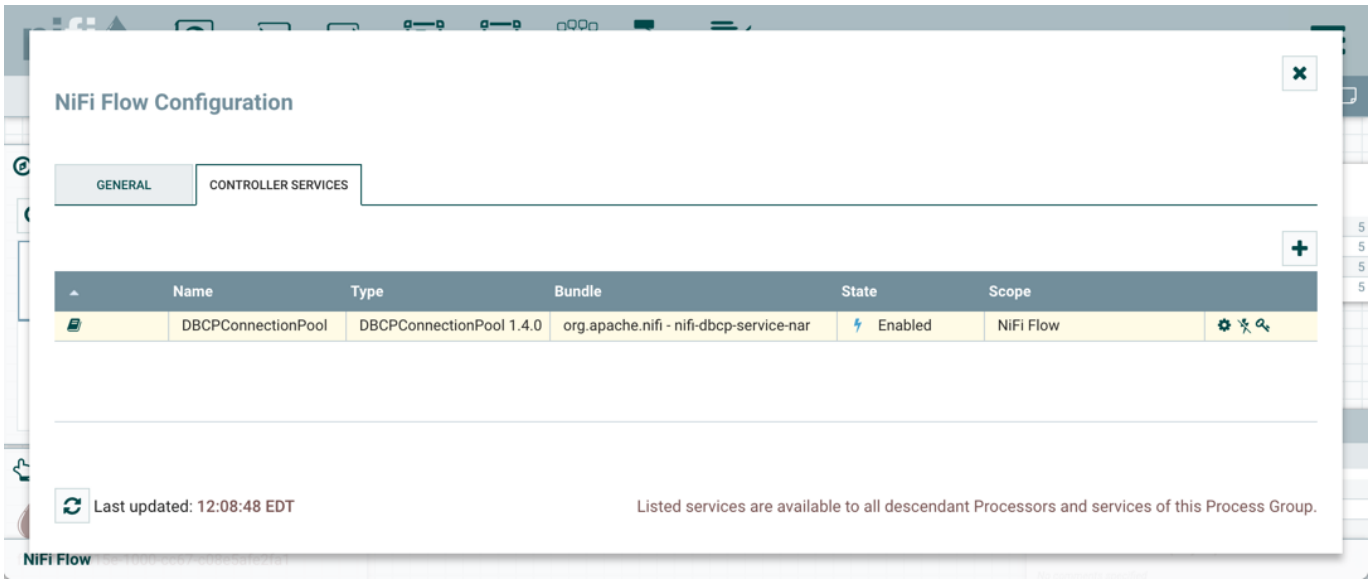

Рис.1.37.: Окно конфигурации Controller Services

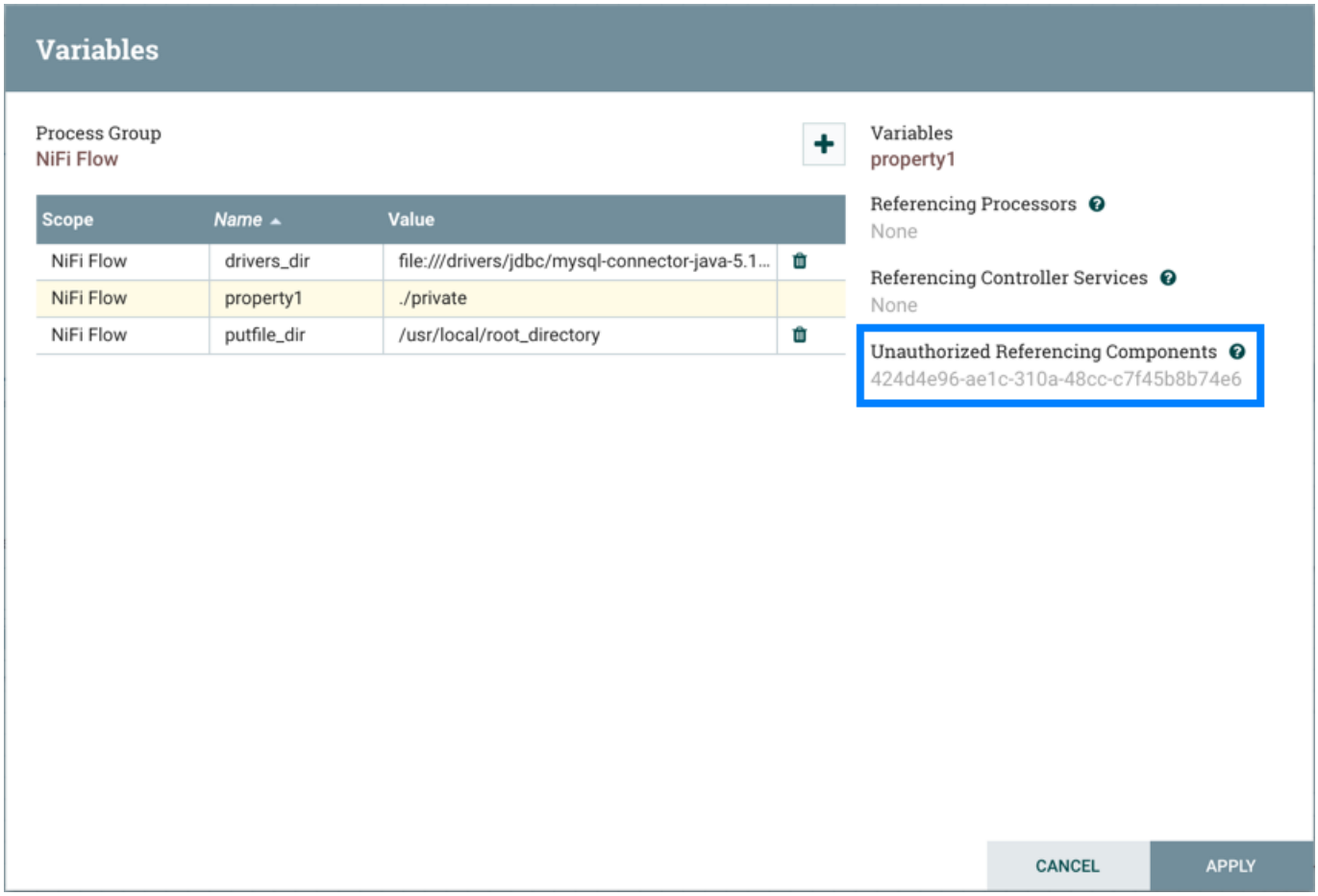

Рис.1.38.: Ссылка на UUID компонента в “Variables”

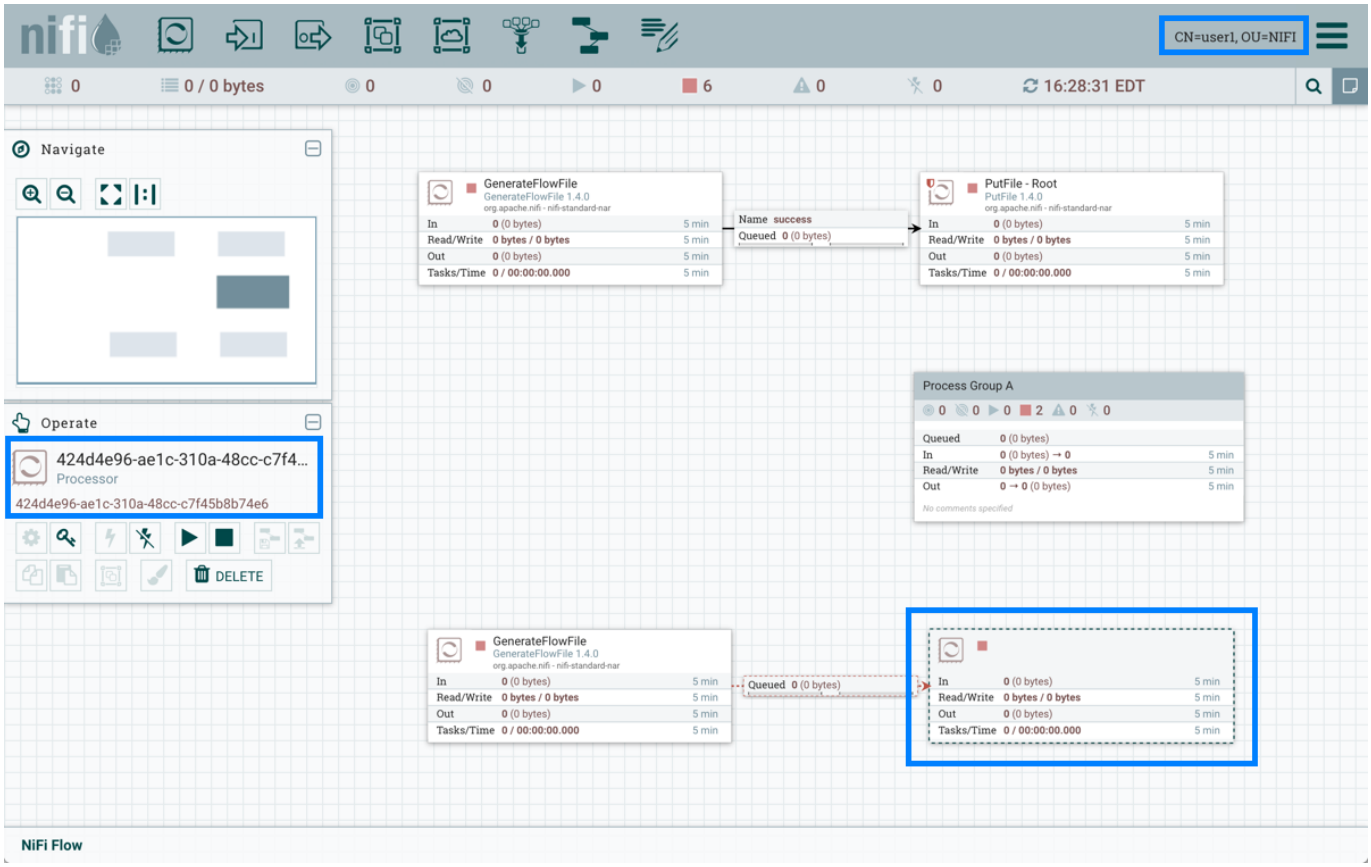

Рис.1.39.: Пример отсутствия прав доступа пользователя к компоненту

Important: Для активации обновлений необходимо перезапустить NiFi

1.8 Управление потоком данных

При добавлении компонента в рабочую область **NiFi** он находится в статусе “Stopped”. Для перевода компонента в рабочее состояние его необходимо запустить. После запуска он может быть остановлен в любой момент времени. В статусе “Stopped” компонент можно настроить, запустить или отключить.

1.8.1 Запуск компонента

Для запуска компонента должны быть выполнены следующие условия:

- Конфигурация компонента действительна;
- Все определенные связи компонента подключены к другим компонентам или настроены на автоматическое завершение;
- Компонент остановлен;
- Компонент доступен;
- Компонент не имеет активных задач.

Для запуска отвечающего требованиям компонента необходимо выбрать его и нажать кнопку “Start” в палитре “Operate Palette” или щелкнув правой кнопкой мыши по компоненту и выбрать “Start” из открывшегося контекстного меню.

При запуске группы процессов запускаются все компоненты группы (включая дочерние), за исключением недействительных или отключенных.

После запуска индикатор состояния Процессора меняется на символ воспроизведения “Play”.

1.8.2 Остановка компонента

Компонент может быть остановлен в любой момент его выполнения. Для этого необходимо щелкнуть правой кнопкой мыши компонент и выбрать команду “Stop” из открывшегося контекстного меню или выбрать команду на панели “Operate Palette”. При остановке группы процессов останавливаются все компоненты группы (включая дочерние).

После остановки индикатор состояния компонента меняется на символ “Stop”.

Остановка компонента не прерывает его текущие задачи, а прекращает планирование выполнения новых задач. Количество активных задач отображается в правом верхнем углу Процессора.

1.8.3 Настройка компонента

Когда компонент доступен, он может быть запущен. При этом пользователи могут отключить компоненты, являющиеся частью потока данных, который все еще собирается. Как правило, если компонент не предназначен для запуска, он отключается, а не остается в состоянии “Stopped”. Это помогает различать компоненты, которые намеренно не запущены, и компоненты, которые могут быть временно остановлены (например, для изменения конфигурации компонента) и случайно не перезапущены.

При необходимости переподключения компонента это можно сделать, выбрав компонент и нажав кнопку “Enable” на панели “Operate Palette”. Это доступно только в том случае, когда выбранный компонент или компоненты отключены. Кроме того, компонент можно включить, установив флажок рядом с параметром “Enabled” на вкладке “Settings” диалогового окна “Processor configuration” или диалогового окна настройки порта.

Индикатор состояния отключенного компонента меняется на “Invalid” или “Stopped” в зависимости от того, является ли компонент действительным.

Компонент отключается путем выбора компонента и нажатия кнопки “Disable” на панели “Operate Palette” или путем снятия флажка рядом с параметром “Enabled” на вкладке “Settings” диалогового окна “Processor configuration” или диалогового окна настройки порта.

Включенными или отключенными компонентами могут быть только порты и процессоры.

1.8.4 Работа удаленных групп процессов

Удаленные группы процессов (Remote Process Group, RPG) предоставляют механизм отправки или извлечения данных из удаленного инстанса **NiFi**. Когда удаленная группа процессов добавляется в рабочую область, она добавляется с отключенной передачей со значком “Transmission Disabled” в верхнем левом углу. Передачу данных можно включить, щелкнув правой кнопкой мыши по RPG и выбрав пункт меню “Enable Transmission”. Это приводит к тому, что все порты, имеющие Соединение, начинают передачу данных, и индикатор состояния меняется на значок “Transmission Enabled”.

При возникновении проблем установления связи с удаленной группой процессов в верхнем левом углу появляется предупреждение “Warning”. Наведение курсора мыши на это предупреждение дает дополнительную информацию о проблеме.

Работа отдельного порта

В некоторых случаях DFM необходимо включить или отключить передачу данных только для определенного порта в группе удаленных процессов. Это можно сделать, щелкнув правой кнопкой мыши на RPG и выбрав пункт меню “Remote ports”. При этом открывается диалоговое окно настроек, в котором доступны конфигурации каждого порта (Рис.1.40.).

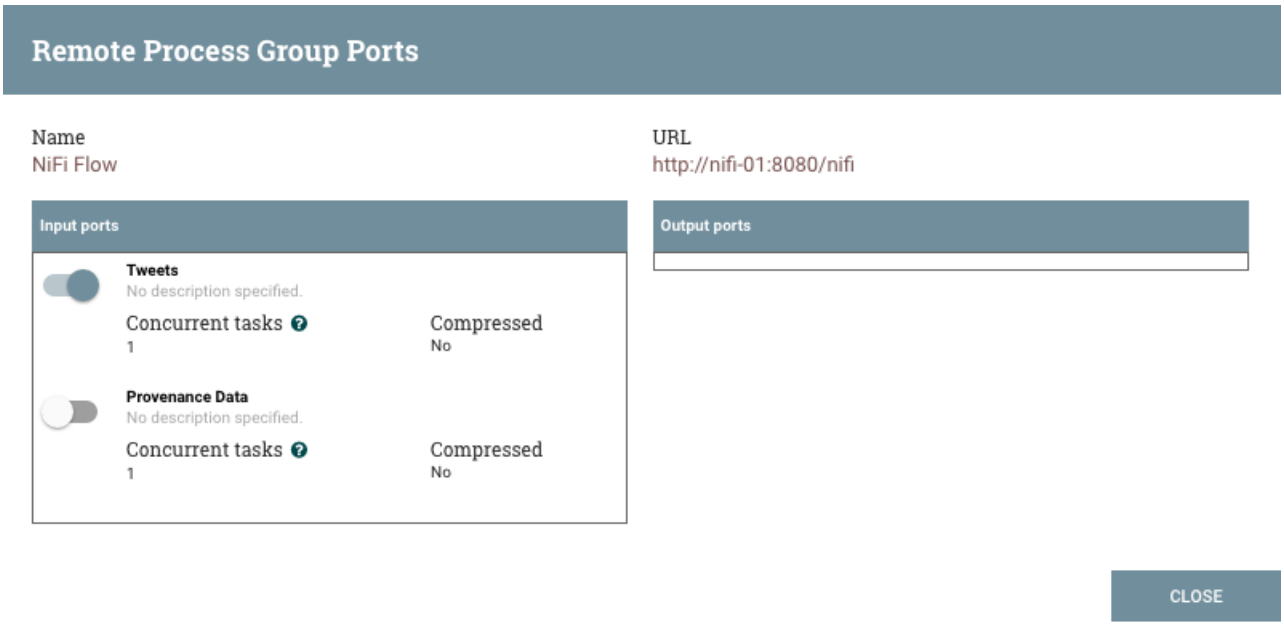

Рис.1.40.: Настройка отдельных портов в RPG

С левой стороны в блоке “Input Ports” перечислены все входные порты, на которые удаленный инстанс **NiFi** позволяет отправлять данные. Справа в блоке “Output Ports” – порты, из которых инстанс может извлекать данные. Если удаленный инстанс использует безопасную связь (его URL-адрес начинается с *https://*, а не *http://*), порты, которые он не сделал доступными, скрыты.

Если порт, который, как ожидается, должен быть показан, но при этом не отображается в диалоговом окне настроек портов, следует убедиться, что у инстанса есть надлежащие разрешения, и что поток в группе удаленных процессов является текущим. Проверить это можно, закрыв диалоговое окно “Port Configuration” и посмотрев в нижний правый угол группы удаленных процессов – отображается дата последнего обновления потока. Если поток является устаревшим, его можно обновить, щелкнув правой кнопкой мыши на RPG и выбрав “Refresh flow”.

Каждый порт отображается с указанием его имени, сопровождающееся описанием, настроенным на данный момент количеством параллельных задач и необходимостью сжатия данных, отправляемых на порт. Слева от информации находится переключатель для включения или выключения порта. Порты, к которым не подключены соединения, выделены серым цветом (Рис.1.41.).

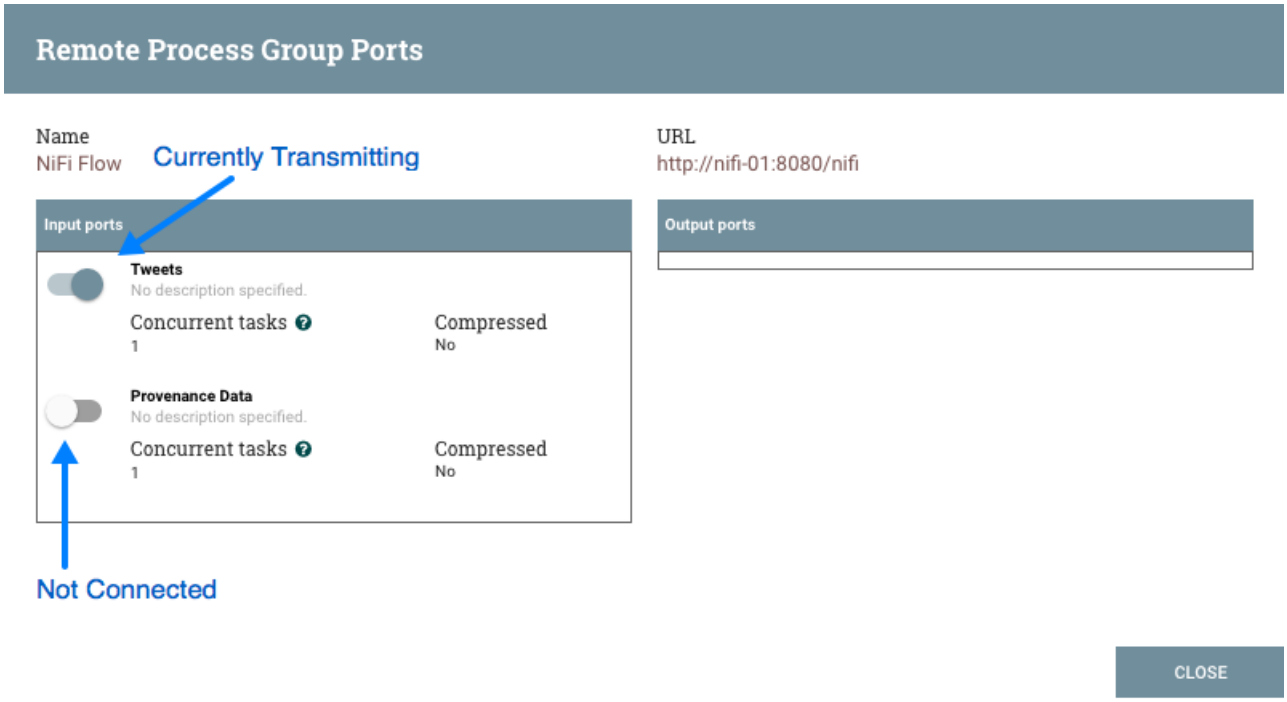

Рис.1.41.: Настройка отдельных портов в RPG

Переключатель предоставляет механизм для включения и отключения передачи данных для каждого порта в удаленной группе процессов независимо друг от друга. Неактивные в данный момент подключенные порты можно настроить, щелкнув значок карандаша “Edit” под переключателем состояния. Это позволяет DFM редактировать количество параллельных задач и определить, следует ли использовать сжатие при передаче данных посредством выбранного порта.

Глава 2

Руководство разработчика приложений для сервиса Kafka

В руководстве приведены сведения необходимые разработчику приложений для сервиса Kafka по работе с платформой ADS.

Руководство может быть полезно разработчикам, администраторам, программистам и сотрудникам подразделений информационных технологий, осуществляющих сопровождение платформы.

Important: Контактная информация службы поддержки – e-mail: info@arenadata.io

2.1 Kafka Clients

Клиенты для работы с Apache Kafka доступны на многих языках программирования (Clients).

В **ADS** предусмотрена возможность установки клиентских библиотек для различных языков, которые обеспечивают как низкоуровневый доступ к **Kafka**, так и потоковую обработку более высокого уровня.

Таблица2.1.: Клиентские библиотки для различных языков

Ссылка на установку	Ссылка на документацию
C/C++	github.com/edenhill/librdkafka
Go	github.com/confluentinc/confluent-kafka-go
Java	Kafka Java Consumer и Kafka Java Producer
JMS	JMS Client
.NET	github.com/confluentinc/confluent-kafka-dotnet
Python	github.com/confluentinc/confluent-kafka-python

В следующей таблице описана поддержка клиента по различным функциям платформы **ADS**.

Таблица2.2.: Поддержка клиента по различным функциям платформы

Функция	C/C++	Go	Java	.NET	Python
Admin API	Да	Да	Да	Нет	Да
Control Center metrics integration	Да	Да	Да	Да	Да
Custom partitioner	Да	Нет	Да	Нет	Нет
Exactly Once Semantics	Нет	Нет	Да	Нет	Нет
Idempotent Producer	Нет	Нет	Да	Нет	Нет
Kafka Streams	Нет	Нет	Да	Нет	Нет
Record Headers	Да	Да	Да	Да	Да
SASL Kerberos/GSSAPI	Да	Да	Да	Да	Да
SASL PLAIN	Да	Да	Да	Да	Да
SASL SCRAM	Да	Да	Да	Да	Да
Simplified installation	Да	Нет	Да	Да	Да
Schema Registry	Да	Нет	Да	Да	Да
Topic Metadata API	Да	Да	Да	Нет	Да

2.1.1 Java

Все JAR-файлы, включенные в пакеты, также доступны в репозитории. Далее приведен пример файла *POM*, показывающий, как добавить репозиторий:

```
<repositories>

  <repository>
    <id>confluent</id>
    <url>https://packages.confluent.io/maven/</url>
  </repository>

  <!-- further repository entries here -->

</repositories>
```

Репозиторий включает в себя скомпилированные версии **Kafka**. Чтобы сослаться на версию **Kafka 2.0** необходимо использовать в файле *pom.xml* следующее:

```
<dependencies>

  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka_2.11</artifactId>
    <version>2.0.1-cp1</version>
  </dependency>

  <!-- further dependency entries here -->

</dependencies>
```

Important: Arenadata всегда вносит исправления в проект с открытым исходным кодом Apache Kafka. Однако точные версии (и их имена), включаемые в платформу ADS, могут отличаться от артефактов Apache при не совпадении релизов. Если они отличаются, Arenadata сохраняет идентификаторы *groupId* и *artifactId*, но добавляет суффикс *-cpX* (где X – цифра) к идентификатору версии ADS для видимого отличия от артефактов Apache

Есть возможность ссылаться на артефакты для всех библиотек **Java**, которые включены в **ADS**. Например, чтобы использовать сериализаторы с открытым исходным кодом **Arenadata**, которые интегрируются с остальной частью платформы **ADS**, необходимо в *pom.xml* включить следующее:

```
<dependencies>

  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <!-- For Confluent Platform 5.0.1 -->
    <version>5.0.1</version>
  </dependency>

  <!-- further dependency entries here -->

</dependencies>
```

2.1.2 C/C++

Клиент **C/C++**, называемый *librdkafka*, доступен в виде исходного кода и в виде предварительно скомпилированных двоичных файлов для дистрибутивов **Linux** на основе **Debian** и **Red Hat**, а также **macOS**. Большинство пользователей используют скомпилированные двоичные файлы.

Для дистрибутивов **Linux** необходимо следовать инструкциям для **Debian** или **Red Hat**, а затем использовать *yum* или *apt-get* для установки соответствующих пакетов. Например, разработчику, создающему приложение **C** на дистрибутиве **Red Hat**, рекомендуется использовать пакет *librdkafka-devel*:

```
sudo yum install librdkafka-devel
```

В дистрибутиве **Debian** используется пакет *librdkafka-dev*:

```
sudo apt-get install librdkafka-dev
```

В **macOS** последняя версия доступна через Homebrew:

```
brew install librdkafka
```

Исходный код доступен в архивах *ZIP* и *TAR* в каталоге *src/*.

2.1.3 JMS

Клиент **JMS** – это библиотека, используемая в приложениях **Java**. Чтобы сослаться на *kafka-jms-client* в проекте для начала необходимо добавить репозиторий в файл *pom.xml*:

```
<repositories>
    <repository>
        <id>confluent</id>
        <url>http://packages.confluent.io/maven/</url>
    </repository>
</repositories>
```

Затем добавить зависимость от клиента **JMS**, а также спецификацию API JMS (при этом заменив *[version]* на требуемую):

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-jms-client</artifactId>
    <version>[version]</version>
</dependency>
<dependency>
```

```
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jms_1.1_spec</artifactId>
    <version>1.1</version>
</dependency>
```

Можно загрузить JAR-файл JMS-клиента напрямую, перейдя по следующему URL-адресу (при этом заменив *[version]* на требуемую):

```
http://packages.confluent.io/maven/io/confluent/kafka-jms-client/[version]/kafka-jms-client
-[version].jar
```

2.1.4 Python

Клиент **Python**, именуемый *confluent-kafka-python*, доступен в PyPI. Клиент **Python** использует *librdkafka* клиента **C**. Поэтому для установки **Python** сначала необходимо инсталлировать C, включая его пакет разработки, а затем установить библиотеку с помощью *pip* (как для **Linux**, так и для **macOS**):

```
pip install confluent-kafka
```

При этом осуществляется глобальная установка пакета для среды **Python**. Для инсталляции клиента только под конкретный проект можно использовать *virtualenv*.

После чего в **Python** можно импортировать библиотеку:

```
from confluent_kafka import Producer

conf = {'bootstrap.servers': 'localhost:9092', 'client.id': 'test', 'default.topic.config': {'acks
↪': 'all'}}
producer = Producer(conf)
producer.produce(topic, key='key', value='value')
```

Исходный код доступен в архивах *ZIP* и *TAR* в каталоге *src/*.

2.1.5 Go

Клиент **Go**, именуемый *confluent-kafka-go*, распространяется через GitHub и gopkg.in с привязкой к конкретным версиям. Клиент **Go** использует *librdkafka* клиента **C** и представляет его как библиотеку **Go**, используя cgo. Для установки клиента **Go** сначала необходимо инсталлировать клиент **C**, включая его пакет разработки, а также набор инструментов для сборки с *pkg-config*. В дистрибутивах **Linux** на основе **Red Hat** в дополнение к *librdkafka* следует установить следующие пакеты:

```
sudo yum groupinstall "Development Tools"
```

В дистрибутивах на основе **Debian**, помимо *librdkafka*, необходимо установить:

```
sudo apt-get install build-essential pkg-config git
```

В **macOS** с помощью Homebrew установить:

```
brew install pkg-config git
```

Далее использовать *go get* для установки библиотеки:

```
go get gopkg.in/confluentinc/confluent-kafka-go.v0/kafka
```

Код **Go** теперь может импортировать и использовать клиент. Также можно собрать и запустить небольшую утилиту командной строки **go-kafkacat**, чтобы убедиться, что установка прошла успешно:

```
go get gopkg.in/confluentinc/confluent-kafka-go.v0/examples/go-kafkacat
$GOPATH/bin/go-kafkacat --help
```

Для настройки статической ссылки к *librdkafka* необходимо добавть флаг *-tags static* к командам *go get*. Это позволяет статически связать саму *librdkafka*, чтобы ее динамическая библиотека не требовалась в целевой системе развертывания. Однако при этом статически связанные зависимости *librdkafka* (такие как *ssl*, *sasl2*, *lz4* и пр.), остаются по-прежнему динамически связанными и требуются в целевой системе. Экспериментальная опция для создания полностью статически связанного двоичного файла также доступна – использование флага *-tags static_all*. При этом требуется, чтобы все зависимости были доступны как статические библиотеки (например, *libsasl2.a*). Статические библиотеки обычно не устанавливаются по умолчанию, но доступны в соответствующих пакетах *-dev* или *-devel* (например, *libsasl2-dev*).

Исходный код доступен в архивах *ZIP* и *TAR* в каталоге *src/*.

2.1.6 .NET

Клиент **.NET**, именуемый *confluent-kafka-dotnet*, доступен в NuGet. Клиент **.NET** использует *librdkafka* клиента **C**. Предварительно скомпилированные двоичные файлы для *librdkafka* предоставляются через зависимый пакет **NuGet** librdkafka.redist для ряда популярных платформ (win-x64, win-x86, debian-x64, rhel-x64 и osx).

Для того, чтобы сослаться на *confluent-kafka-dotnet* из проекта, необходимо выполнить следующую команду в консоли диспетчера пакетов:

```
PM> Install-Package Confluent.Kafka
```

Important: Зависимый пакет *librdkafka.redist* устанавливается автоматически

Для того, чтобы сослаться на *confluent-kafka-dotnet* в файле *project.json*, необходимо включить следующую ссылку в раздел зависимостей:

```
"dependencies": {
    ...
    "Confluent.Kafka": "0.9.4"
    ...
}
```

И затем выполнить команду `dotnet restore`, чтобы восстановить зависимости проекта через **NuGet**.

Клиент *confluent-kafka-dotnet* предназначен для платформ **net451** и **netstandard1.3** и поддерживается в **.NET Framework** версии *4.5.1* и выше и **.NET Core** версии *1.0* (в **Windows**, **Linux** и **Mac**) и выше. Не поддерживается на **Mono**.

2.2 Kafka Java Consumer

Платформа **ADS** включает в себя Java consumer, поставляемый вместе с **Kafka**.

В документе представлен общий обзор работы потребителя, введение в параметры конфигурации для настройки и примеры из каждой клиентской библиотеки.

2.2.1 Концепция

Consumer group – группа потребителей, взаимодействующих для использования данных из топиков. Партиции в топиках делятся между потребителями в группе, и при изменениях в группе партиции перераспределяются таким образом, что каждый потребитель получает пропорциональную долю партиций. Такой процесс называется перебалансировкой группы (rebalancing the group).

Основное различие в управлении группами между старым “high-level” потребителем и новым заключается в том, что первый зависит от **ZooKeeper**, а второй использует групповой протокол, встроенный в саму

Kafka. В данном протоколе один из брокеров назначается координатором группы и отвечает за управление ее потребителями и за назначение им партиций.

Координатор каждой группы выбирается из лидеров внутренних смещений *__consumer_offsets*. Обычно идентификатор группы хэшируется в одной из партиций топика, и лидер данной партиции выбирается в качестве координатора. Таким образом, управление группами потребителей разделяется примерно поровну между всеми брокерами в кластере, что позволяет масштабировать количество групп за счет увеличения числом брокеров.

Когда потребитель запускается, он находит координатора для своей группы и отправляет запрос на присоединение. При этом координатор начинает перебалансировку, что приводит к формированию новой группы.

Каждый участник в группе должен посылать heartbeat-сообщения координатору. В случае если до истечения настроенного тайм-аута сессии такового не получено, координатор исключает потребителя из группы и переназначает его партиции.

Управление смещением (Offset Management)

После получения потребителем назначения от координатора необходимо выявить начальную позицию для каждой определенной партиции. Когда группа создается впервые, до того, как какие-либо сообщения были использованы, позиция устанавливается в соответствии с политикой сброса смещения (*auto.offset.reset*). Как правило, потребление начинается с самого раннего либо с самого позднего смещения.

Потребителю необходимо фиксировать свои смещения в партиции в соответствии с ходом прочтения сообщений. Поскольку, если потребитель выходит из строя или выключается, его партиции переназначаются другому участнику группы, который начинает потребление сообщений с последнего закоммиченного смещения. В случае аварийного завершения работы потребитиеля до того, как какое-либо его смещение зафиксировано, следующий потребитель использует политику сброса.

Политика фиксации смещения имеет ключевое значение для обеспечения необходимых приложению гарантий доставки сообщений. По умолчанию потребитель настроен на использование политики автоматического коммита, которая инициирует фиксацию с периодическим интервалом. Также потребителем поддерживается API, который можно использовать для ручного управления смещением. В примерах приведено несколько подробных случаев API-фиксации и обсуждение компромиссов с точки зрения производительности и надежности (*Примеры*).

При записи во внешнюю систему позиция потребителя должна быть согласована с тем, что хранится в виде выходных данных. Именно поэтому потребитель хранит свое смещение в том же месте, где выходные данные. Например, Kafka Connect записывает данные в **HDFS** вместе со смещениями считываемых данных, что гарантирует обоюдное обновление данных и смещений. Аналогичная схема применяется для многих других систем данных, требующих более строгой семантики, и для которых сообщения не имеют первичного ключа для обеспечения дедупликации.

Так **Kafka** поддерживает обработку exactly-once в Kafka Streams, и поставщик или потребитель транзакций может использоваться для обеспечения доставки exactly-once при передаче и обработке данных между топиками **Kafka**. В противном случае **Kafka** гарантирует доставку at-least-once по умолчанию, но при этом можно реализовать доставку at-most-once, отключив повторные попытки для поставщика и зафиксировав смещения в потребителе перед обработкой пакета сообщений.

2.2.2 Конфигурация

Полный список параметров конфигурации доступен в документе Настройки платформы Arenadata Streaming. Но некоторые из ключевых параметров и их влияние на поведение потребителя описаны в текущей главе.

Базовая конфигурация (Core Configuration)

Единственная обязательная настройка – это *bootstrap.servers*, но при этом должен быть установлен *client.id* для сопоставления запросов в брокере со сделавшим их инстансом клиента. Как правило, все потребители в одной группе используют один и тот же идентификатор клиента.

Конфигурация группы (Group Configuration)

Параметр *group.id* должен быть всегда настроен за исключением случаев, когда API используется просто по назначению и нет необходимости хранить смещения в **Kafka**.

Тайм-аутом сессии можно управлять в параметре *session.timeout.ms*. Значение по умолчанию установлено на *30 секунд*, но если приложению требуется больше времени для обработки сообщений, то для избежания чрезмерной перебалансировки значение параметра можно безопасно увеличить. Это в основном актуально при использовании Java consumer и обработке сообщений в одном потоке. В таком случае также можно регулировать параметр *max.poll.records* для настройки количества требуемых для обработки на каждой итерации цикла записей (более подробно вопрос рассмотрен в главе *Основные возможности*).

Основным недостатком применения долгого тайм-аута сессии является то, что координатору требуется больше времени для обнаружения сбоя инстанса потребителя, а это значит, что другому потребителю в группе требуется больше времени для передачи партиций. Но при этом в случае необходимости нормального выключения потребитель отправляет координатору явный запрос покинуть группу, который инициирует немедленную перебалансировку.

Другим параметром, влияющим на поведение перебалансировки, является *heartbeat.interval.ms*. Он контролирует, как часто потребитель должен отправлять heartbeats-сообщения координатору. Это также способ, когда необходимость перебалансировки определяется засчет потребителя, поэтому более короткий интервал heartbeats-сообщений обычно означает более быструю перебалансировку. Значение по умолчанию составляет *3 секунды*. Для больших групп целесообразно увеличить значение параметра.

Управление смещением (Offset Management)

Двумя основными параметрами, влияющими на управление смещением, являются автоматическая фиксация и политика сброса смещения. В первом случае при установленном по умолчанию параметре *enable.auto.commit* потребитель автоматически фиксирует смещения с заданным в *auto.commit.interval.ms* интервалом (по умолчанию – *5 секунд*).

Второй параметр *auto.offset.reset* определяет поведение потребителя, когда нет зафиксированной позиции смещения (в случае, когда группа инициализируется впервые), или когда оно выходит за пределы диапазона. Можно установить сброс положения на самое раннее смещение – *earliest*, либо на самое позднее – *latest* (задано по умолчанию). Также можно выбрать значение *none* для самостоятельной установки начального смещения и ручной обработки ошибок вне диапазона.

2.2.3 Инициализация

Потребитель Java consumer создается с помощью стандартного файла свойств *Properties*:

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("group.id", "foo");
config.put("bootstrap.servers", "host1:9092,host2:9092");
new KafkaConsumer<K, V>(config);
```

Important: Ошибки конфигурации приводят к возникновению исключения *KafkaException* из конструктора *KafkaConsumer*

Конфигурация **C/C++** (*librdkafka*) похожа, но при этом необходимо обрабатывать ошибки конфигурации непосредственно при настройке свойств:

```
char hostname[128];
char errstr[512];

rd_kafka_conf_t *conf = rd_kafka_conf_new();

if (gethostname(hostname, sizeof(hostname))) {
 fprintf(stderr, "%% Failed to lookup hostname\n");
 exit(1);
}

if (rd_kafka_conf_set(conf, "client.id", hostname,
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
 fprintf(stderr, "%% %s\n", errstr);
 exit(1);
}

if (rd_kafka_conf_set(conf, "group.id", "foo",
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
 fprintf(stderr, "%% %s\n", errstr);
 exit(1);
}

if (rd_kafka_conf_set(conf, "bootstrap.servers", "host1:9092,host2:9092",
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
 fprintf(stderr, "%% %s\n", errstr);
 exit(1);
}

/* Create Kafka consumer handle */
rd_kafka_t *rk;
if (!(rk = rd_kafka_new(RD_KAFKA_CONSUMER, conf,
                        errstr, sizeof(errstr)))) {
 fprintf(stderr, "%% Failed to create new consumer: %s\n", errstr);
 exit(1);
}
```

Клиент **Python** может быть настроен через словарь следующим образом:

```
from confluent_kafka import Consumer

conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'group.id': "foo",
        'default.topic.config': {'auto.offset.reset': 'smallest'}}

consumer = Consumer(conf)
```

Клиент **Go** использует объект *ConfigMap* для передачи конфигурации потребителю:

```
import (
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

consumer, err := kafka.NewConsumer(&kafka.ConfigMap{
     "bootstrap.servers":    "host1:9092,host2:9092",
     "group.id":             "foo",
     "default.topic.config": kafka.ConfigMap{"auto.offset.reset": "smallest"}})
```

В **C#** используется *Dictionary<string, object>*:

```
using System.Collections.Generic;
using Confluent.Kafka;

...

var config = new Dictionary<string, object>
{
    { "bootstrap.servers", "host1:9092,host2:9092" },
    { "group.id", "foo" },
    { "default.topic.config", new Dictionary<string, object>
        {
            { "auto.offset.reset", "smallest" }
        }
    }
}

using (var consumer = new Consumer<Null, string>(config, null, new StringDeserializer(Encoding.
↪UTF8)))
{
    ...
}
```

2.2.4 Основные возможности

Хотя Java-клиент и *librdkafka* имеют много общих опций конфигурации и базовых функций, они используют довольно разные подходы, когда дело доходит до их потоковой модели и работы с потребителями. Прежде чем углубляться в примеры, полезно разобраться в дизайне API каждого клиента.

Java Client

Java Client разработан вокруг цикла обработки событий под управляением *poll()* API. Конструкция мотивирована системными вызовами UNIX *select* и *poll*. Базовый цикл потребления с Java API обычно принимает следующую форму:

```
while (running) {
  ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
  process(records); // application-specific processing
  consumer.commitSync();
}
```

В Java consumer нет фонового потока. API зависит от вызовов *poll()* для управления всеми операциями ввода-вывода, включая:

- Присоединение к группе потребителей и обработка перебалансировкой партиций;
- Периодичная отправка heartbeats-сообщений;
- Периодичная отправка зафиксированных смещений (при включенном автокоммите);
- Отправка и получение запросов на выборку для назначенных партиций.

Такая однопоточная модель означает, что нельзя отправлять heartbeats-сообщения, пока приложение обрабатывает записи по вызову *poll()*. Это приводит к тому, что потребитель выпадает из группы, если цикл обработки событий завершается, либо если задержка в обработке записи приводит к истечению времени ожидания сессии до следующей итерации цикла. Так и было задумано. Одна из проблем, которую пытается

решить **Java Client**, – обеспечение жизнедеятельности потребителей в группе. В то время, пока потребителю назначены партиции, другие члены группы не могут их же использовать, поэтому важно убедиться, что каждый конкретный потребитель действительно прогрессирует.

Данная функция защищает приложение от большого класса сбоев, но недостатком является то, что необходима настройка времени ожидания сессии так, чтобы потребитель не превышал его в своей обычной обработке записей. Кроме этого параметр *max.poll.records* устанавливает верхнюю границу количества записей, возвращаемых при каждом вызове. Поэтому важно использовать *poll()* и *max.poll.records* с достаточно большим тайм-аутом сессии (например, от *30* до *60 секунд*) и ограничивать количество обработанных записей на каждой итерации.

В случае если данные параметры не настроены надлежащим образом, это, как правило, приводит к исключению *CommitFailedException* смещения для обработанных записей. При использовании политики автоматической фиксации, можно даже не заметить, когда это происходит, так как потребитель молча игнорирует сбои коммитов (если только это не происходит достаточно часто, чтобы повлиять на показатели задержки). Исключение можно перехватить и либо проигнорировать, либо выполнить необходимую логику отката:

```
while (running) {
  ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
  process(records); // application-specific processing
  try {
    consumer.commitSync();
  } catch (CommitFailedException e) {
    // application-specific rollback of processed records
  }
}
```

C/C++ Client (librdkafka)

Librdkafka использует многопоточный подход к потреблению **Kafka**. С точки зрения пользователя, взаимодействие с API не слишком отличается от примера, используемого Java-клиентом, когда пользователь вызывает *rd_kafka_consumer_poll* в цикле, хотя данный API возвращает только одно сообщение или событие за раз:

```
while (running) {
 rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
 if (rkmessage) {
   msg_process(rkmessage);
   rd_kafka_message_destroy(rkmessage);

   if ((++msg_count % MIN_COMMIT_COUNT) == 0)
     rd_kafka_commit(rk, NULL, 0);
 }
}
```

В отличие от Java-клиента, *librdkafka* выполняет всю выборку и координирует взаимодействие в фоновых потоках, что освобождает от сложности настройки тайм-аута сессии в соответствии с ожидаемым временем обработки. Однако, поскольку фоновый поток поддерживает потребителя до тех пор, пока клиент не закроется, важно убедиться, что процесс не простаивает, так как в этом случае он продолжает удерживать назначенные ему партиции.

При этом перебалансировка партиций также происходит в фоновом потоке, а это говорит о том, что все равно приходится обрабатывать потенциальные ошибки коммита, поскольку потребитель может больше не иметь того же назначения партиций, когда начинается фиксация. Это не требуется при включенном автокоммите, так как при этом ошибки коммита игнорируются в автоматическом режиме, но это также означает, что нет возможности откатить обработку.

```
while (running) {
  rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 1000);
  if (!rkmessage)
    continue; // timeout: no message

  msg_process(rkmessage); // application-specific processing
  rd_kafka_message_destroy(rkmessage);

  if ((++msg_count % MIN_COMMIT_COUNT) == 0) {
    rd_kafka_resp_err_t err = rd_kafka_commit(rk, NULL, 0);
    if (err) {
      // application-specific rollback of processed records
    }
  }
}
```

Python, Go и .NET Clients

Клиенты **Python**, **Go** и **.NET** на внутреннем уровне используют *librdkafka*, поэтому у них также применяется многопоточный подход к потреблению **Kafka**. С точки зрения пользователя, взаимодействие с API не слишком отличается от примера, используемого Java-клиентом, когда пользователь вызывает метод *poll()* в цикле, хотя данный API возвращает только одно сообщение за раз.

Python:

```
try:
    msg_count = 0
    while running:
        msg = consumer.poll(timeout=1.0)
        if msg is None: continue

        msg_process(msg) # application-specific processing
        msg_count += 1
        if msg_count % MIN_COMMIT_COUNT == 0:
            consumer.commit(async=False)
finally:
    # Shut down consumer
    consumer.close()
```

Go:

```
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        // application-specific processing
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

Поведение потребителя **C#** аналогично, за исключением того, что перед входом в цикл *Poll* необходимо настроить обработчики для различных типов событий, что эффективно делается внутри метода *Poll* (важно обратить внимание, что весь код выполняется в том же потоке):

```
consumer.OnMessage += (_, msg) =>
{
    // handle message.
}

consumer.OnPartitionEOF += (_, end)
    => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

consumer.OnError += (_, error)
{
    Console.WriteLine($"Error: {error}");
    cancelled = true;
}

while (!cancelled)
{
    consumer.Poll(TimeSpan.FromSeconds(1));
}
```

2.2.5 Примеры

Далее приведены подробные примеры использования consumer API с особым вниманием к управлению смещением и семантике доставки.

Basic Poll Loop

API потребителя сосредоточен вокруг метода *poll()* для получения записей от брокеров и метода *subscribe()* для выбора топиков. Как правило, потребитель первоначально обращается к методу *subscribe()* для настройки интересующих топиков, а затем запускает цикл *poll()* до завершения работы приложения.

Потребитель намеренно избегает конкретной модели потоков, так как это не безопасно для многопоточного доступа и не дает возможности наличия собственных фоновых потоков. В частности, это означает, что все операции ввода-вывода происходят в потоке, вызыванном методом *poll()*. В приведенном ниже примере цикл опроса заключен в *Runnable*, который упрощает использование с *ExecutorService*:

```
public abstract class BasicConsumeLoop implements Runnable {
  private final KafkaConsumer<K, V> consumer;
  private final List<String> topics;
  private final AtomicBoolean shutdown;
  private final CountDownLatch shutdownLatch;

  public BasicConsumeLoop(Properties config, List<String> topics) {
    this.consumer = new KafkaConsumer<>(config);
    this.topics = topics;
    this.shutdown = new AtomicBoolean(false);
    this.shutdownLatch = new CountDownLatch(1);
  }

  public abstract void process(ConsumerRecord<K, V> record);

  public void run() {
    try {
      consumer.subscribe(topics);

      while (!shutdown.get()) {
        ConsumerRecords<K, V> records = consumer.poll(500);
        records.forEach(record -> process(record));
```

```
      }
    } finally {
      consumer.close();
      shutdownLatch.countDown();
    }
  }

  public void shutdown() throws InterruptedException {
    shutdown.set(true);
    shutdownLatch.await();
  }
}
```

В примере жестко запрограммировано время ожидания опроса на *500 миллисекунд*, то есть, если никаких записей не получено до истечения тайм-аута, *poll()* возвращает пустой набор записей. В случае если обработка сообщений связана с дополнительными затратами на настройку, можно добавить проверку ярлыков.

Для отключения потребителя добавляется флаг, который проверяется на каждой итерации цикла. При этом потребитель ожидает *500 миллисекунд* (плюс время обработки сообщения) перед завершением работы. Лучший подход представлен далее в примере.

Важно обратить внимание, что всегда следует вызывать *close()* после завершения работы потребителя. Это обеспечивает закрытие активных сокетов и очистку внутреннего состояния. Также это немедленно инициирует перебалансировку группы, что в свою очередь гарантирует переназначение всех принадлежащих данному потребителю партиций другому члену группы. Если не выполнить закрытие должным образом, брокер инициирует перебалансировку только после истечения времени ожидания сессии. В примере добавлена защелка (latch) для того, чтобы у потребителя было время завершить закрытие перед выключением.

Этот же пример выглядит аналогично в *librdkafka*:

```
static int shutdown = 0;
static void msg_process(rd_kafka_message_t message);

void basic_consume_loop(rd_kafka_t *rk,
                        rd_kafka_topic_partition_list_t *topics) {
  rd_kafka_resp_err_t err;

  if ((err = rd_kafka_subscribe(rk, topics))) {
    fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
    exit(1);
  }

  while (running) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
    if (rkmessage) {
      msg_process(rkmessage);
      rd_kafka_message_destroy(rkmessage);
    }
  }

  err = rd_kafka_consumer_close(rk);
  if (err)
    fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
  else
    fprintf(stderr, "%% Consumer closed\n");
}
```

В **Python**:

```
running = True

def basic_consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
    finally:
        # Close down consumer to commit final offsets.
        consumer.close()

def shutdown():
    running = False
```

В **Go**:

```
err = consumer.SubscribeTopics(topics, nil)

for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        fmt.Printf("%% Message on %s:\n%s\n",
            e.TopicPartition, string(e.Value))
    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}

consumer.Close()
```

В **C#**:

```
using (var consumer = new Consumer<Null, string>(config, null, new StringDeserializer(Encoding.
↪UTF8)))
{
    consumer.OnMessage += (_, msg)
        => Console.WriteLine($"Message value: {msg.Value}");

    consumer.OnPartitionEOF += (_, end)
        => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");
```

```
    consumer.OnError += (_, error)
    {
        Console.WriteLine($"Error: {error}");
        cancelled = true;
    }

    consumer.Subscribe(topics);

    while (!cancelled)
    {
        consumer.Poll(TimeSpan.FromSeconds(1));
    }
}
```

Хотя API-интерфейсы схожи, клиенты **C/C++**, **Python**, **Go** и **C#** используют другой подход, нежели **Java**. В то время как потребитель **Java** выполняет все операции ввода-вывода и обработку в потоке переднего плана (foreground thread), остальные клиенты используют фоновый поток (background thread). Основным следствием использования многопоточности (multiple threads) является то, что вызов *rd_kafka_consumer_poll* или *Consumer.poll()* абсолютно безопасен, то есть можно распараллеливать обработку сообщений по нескольким потокам. С высокого уровня опрос извлекает сообщения из очереди, которая заполняется в фоновом потоке.

Другим следствием использования фонового потока является то, что в нем выполняются все heartbeats-сообщения и перебалансировки. Преимущество заключается в отсутствии беспокойства об обработке сообщений, которая может стать следствием "пропуска" потребителем перебалансировки. Недостатком является то, что фоновый поток продолжает отправку heartbeats-сообщений, даже если процессор сообщений умер. А в таком случае потребитель удерживает свои партиции, и задержка чтения продолжается до выключения процесса.

Хотя клиенты используют различные подходы, они не так далеки друг от друга, как кажется. Для обеспечения той же абстракции в клиенте **Java** можно поместить очередь между циклом опроса и процессором сообщений, тогда poll loop заполняет очередь, а процессоры по ней извлекают сообщения.

Shutdown и Wakeup

Альтернативным шаблоном для цикла опроса в Java-клиенте является использование *Long.MAX_VALUE* для тайм-аута. Для выхода из цикла можно использовать метод потребителя *wakeup()* из отдельного потока. Это вызывает исключение *WakeupException* из потока, блокирующего *poll()*. Если поток блокируется некорректно, то это приводит к вызову следующего опроса.

```
public abstract class ConsumeLoop implements Runnable {
  private final KafkaConsumer<K, V> consumer;
  private final List<String> topics;
  private final CountDownLatch shutdownLatch;

  public BasicConsumeLoop(KafkaConsumer<K, V> consumer, List<String> topics) {
    this.consumer = consumer;
    this.topics = topics;
    this.shutdownLatch = new CountDownLatch(1);
  }

  public abstract void process(ConsumerRecord<K, V> record);

  public void run() {
    try {
      consumer.subscribe(topics);
```

```
      while (true) {
        ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
        records.forEach(record -> process(record));
      }
    } catch (WakeupException e) {
      // ignore, we're closing
    } catch (Exception e) {
      log.error("Unexpected error", e);
    } finally {
      consumer.close();
      shutdownLatch.countDown();
    }
  }

  public void shutdown() throws InterruptedException {
    consumer.wakeup();
    shutdownLatch.await();
  }
}
```

Синхронные коммиты

В предыдущих примерах предполагается, что потребитель настроен на автоматическую фиксацию смещений (по умолчанию). Auto-commit в основном работает как cron с периодом, установленным через свойство конфигурации *auto.commit.interval.ms*. Если потребитель аварийно завершает работу, то после перезапуска или перебалансировки положение всех принадлежащих ему партиций сбрасывается до последнего зафиксированного смещения. При этом последний коммит может быть таким же старым, как и сам интервал автоматической фиксации. Любые сообщения, поступившие с момента последнего коммита, необходимо прочитать повторно.

Очевидно, что для сокращения окна дубликатов можно уменьшить интервал автоматической фиксации, но некоторым пользователям может потребоваться еще более точный контроль над смещениями. Поэтому потребитель поддерживает commit API, который дает полный контроль над смещениями. Самый простой и надежный способ ручной фиксации смещений – использовать синхронную фиксацию с помощью *commitSync()*, вызов которой блокирует поток до успешно выполненного коммита.

При непосредственном использовании API фиксации необходимо сначала отключить автоматический коммит в конфигурации, установив для свойства *enable.auto.commit* значение *false*.

```
private void doCommitSync() {
  try {
    consumer.commitSync();
  } catch (WakeupException e) {
    // we're shutting down, but finish the commit first and then
    // rethrow the exception so that the main loop can exit
    doCommitSync();
    throw e;
  } catch (CommitFailedException e) {
    // the commit failed with an unrecoverable error. if there is any
    // internal state which depended on the commit, you can clean it
    // up here. otherwise it's reasonable to ignore the error and go on
    log.debug("Commit failed", e);
  }
}

public void run() {
  try {
    consumer.subscribe(topics);
```

```
    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      records.forEach(record -> process(record));
      doCommitSync();
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
    shutdownLatch.countDown();
  }
}
```

В данном примере блок *try/catch* добавлен к вызову *commitSync*. Когда фиксация не может быть завершена по причине перебалансировки группы, выдается *CommitFailedException*. Это главное, что с осторожностью необходимо соблюдать при использовании клиента **Java**. Поскольку все сетевые операции ввода-вывода (включая heartbeats-сообщения) и обработка сообщений выполняются в потоке переднего плана, тайм-аут сессии может истечь во время обработки пакета сообщений. Чтобы справиться с этим, есть два варианта.

В первом варианте сначала можно настроить параметр *session.timeout.ms*, чтобы у обработчика было достаточно времени для завершения обработки сообщений. Затем можно настроить *max.partition.fetch.bytes*, чтобы ограничить объем данных, возвращаемых в одном пакете, но при этом приходится учитывать, сколько партиций содержится в подписанных топиках.

Второй вариант заключается в обработке сообщений в отдельном потоке, но тогда приходится управлять потоком передачи данных, чтобы потоки не отставали. Например, простого помещения сообщений в очередь блокировки, вероятно, недостаточно, если скорость обработки не поспевает за скоростью доставки (в этом случае может не понадобиться отдельный поток). Это может даже усугубить проблему, если цикл опроса заблокирован при вызове метода *offer()*, так как тогда фоновый поток обрабатывает еще больший пакет сообщений. API **Java** предлагает метод *pause()*, чтобы помочь в подобных ситуациях.

В данном случае необходимо установить *session.timeout.ms* достаточно большим, чтобы сбои при перебалансировках происходили реже. Как упомянуто выше, единственным недостатком этого является более длительная задержка переназначения партиций в случае серьезного сбоя (когда потребитель не может быть чисто завершен с помощью *close()*), что на практике редко происходит.

Важно проявить осторожность, так как функция *wakeup()* может быть запущена, пока коммит находится в состоянии ожидания. Рекурсивный вызов безопасней, поскольку инициирует wakeup только один раз.

В **C/C++** (*librdkafka*) можно получить похожее поведение с *rd_kafka_commit*, который используется как для синхронных, так и для асинхронных фиксаций. Однако подход немного отличается, поскольку *rd_kafka_consumer_poll* возвращает отдельные сообщения вместо пакетов, как это делает потребитель **Java**.

```
void consume_loop(rd_kafka_t *rk,
                  rd_kafka_topic_partition_list_t *topics) {
  static const int MIN_COMMIT_COUNT = 1000;

  int msg_count = 0;
  rd_kafka_resp_err_t err;

  if ((err = rd_kafka_subscribe(rk, topics))) {
    fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
    exit(1);
  }
```

```
  while (running) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
    if (rkmessage) {
      msg_process(rkmessage);
      rd_kafka_message_destroy(rkmessage);

      if ((++msg_count % MIN_COMMIT_COUNT) == 0)
        rd_kafka_commit(rk, NULL, 0);
    }
  }

  err = rd_kafka_consumer_close(rk);
  if (err)
    fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
  else
    fprintf(stderr, "%% Consumer closed\n");
}
```

В данном примере синхронная фиксация запускается каждые *1000* сообщений. Вторым аргументом *rd_kafka_commit* является список смещений, которые должны быть зафиксированы; при значении *NULL* *librdkafka* фиксирует последние смещения для назначенных позиций. Третий аргумент в *rd_kafka_commit* – флаг, который определяет асинхронность вызова. Коммит также можно активировать по истечению тайм-аута, чтобы убедиться, что зафиксированная позиция регулярно обновляется.

Поскольку клиент **Python** внутренне использует *librdkafka*, он применяет аналогичный шаблон, устанавливая параметр *async* для вызова метода *Consumer.commit()*. Этот метод также может принимать взаимоисключающие смещения параметров ключевых слов для явного перечисления смещений каждой назначенной партиции топика и *message*, которые фиксируют смещения относительно объекта *Message*, возвращаемого функцией *poll()*.

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(async=False)
    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```

Клиент **Go** также внутренне использует *librdkafka*, поэтому он применяет похожий шаблон, но обеспечивает при этом только синхронный вызов метода *Commit()*. Другие варианты методов

фиксации также принимают список смещений для коммитов или *Message*, чтобы зафиксировать смещения относительно считываемого сообщения. При использовании ручного коммита важно отключить конфигурацию *enable.auto.commit*.

```
msg_count := 0
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        msg_count += 1
        if msg_count % MIN_COMMIT_COUNT == 0 {
            consumer.Commit()
        }
        fmt.Printf("%% Message on %s:\n%s\n",
            e.TopicPartition, string(e.Value))

    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

Клиент **C#** обеспечивает метод *CommitAsync* с возможными перегрузками. Его можно использовать синхронно, отвечая *Result* или *Wait()* на возвращаемый *Task*. Существуют варианты, которые фиксируют все смещения в текущем назначении, конкретный список смещений или смещение на основе *Message*.

```
var msgCount = 0;

consumer.OnMessage += (_, msg) =>
{
    msgCount += 1;
    if (msgCount % MIN_COMMIT_COUNT == 0)
    {
        consumer.CommitAsync().Wait();
    }
    Console.WriteLine($"Message value: {msg.Value}");
}

consumer.OnPartitionEOF += (_, end)
    => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

consumer.OnError += (_, error)
{
    Console.WriteLine($"Error: {error}");
    cancelled = true;
}

while (!cancelled)
{
    consumer.Poll(TimeSpan.FromSeconds(1));
}
```

Использование автоматической фиксации обеспечивает доставку “at least once”: **Kafka** гарантирует, что ни одно сообщение не будет пропущено, но возможны дубликаты. В предыдущем примере обеспечивается такая

доставка, поскольку фиксация следует за обработкой сообщения. Однако, изменив запрос, можно получить доставку “at most once”. Но при этом следует быть осторожнее с ошибкой коммита, для этого необходимо изменить *doCommitSync*, чтобы он возвращал информацию об успешности транзакции. Так же при синхронной фиксации отменяется необходимость в перехвате исключения *WakeupException*.

```
private boolean doCommitSync() {
  try {
    consumer.commitSync();
    return true;
  } catch (CommitFailedException e) {
    // the commit failed with an unrecoverable error. if there is any
    // internal state which depended on the commit, you can clean it
    // up here. otherwise it's reasonable to ignore the error and go on
    log.debug("Commit failed", e);
    return false;
  }
}

public void run() {
  try {
    consumer.subscribe(topics);

    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      if (doCommitSync())
        records.forEach(record -> process(record));
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
    shutdownLatch.countDown();
  }
}
```

C/C++ (*librdkafka*):

```
void consume_loop(rd_kafka_t *rk,
                  rd_kafka_topic_partition_list_t *topics) {
  rd_kafka_resp_err_t err;

  if ((err = rd_kafka_subscribe(rk, topics))) {
    fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
    exit(1);
  }

  while (running) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
    if (rkmessage && !rd_kafka_commit_message(rk, rkmessage, 0)) {
      msg_process(rkmessage);
      rd_kafka_message_destroy(rkmessage);
    }
  }

  err = rd_kafka_consumer_close(rk);
  if (err)
```

```
    fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
  else
    fprintf(stderr, "%% Consumer closed\n");
}
```

Python:

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
            else:
                consumer.commit(async=False)
                msg_process(msg)

    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```

Go:

```
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        err = consumer.CommitMessage(e)
        if err == nil {
            msg_process(e)
        }

    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

C#:

```
consumer.OnMessage += (_, msg) =>
{
    var err = consumer.CommitAsync().Result.Error;
    if (!err)
    {
```

```
        processMessage(msg);
    }
}

consumer.OnPartitionEOF += (_, end)
    => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

consumer.OnError += (_, error)
{
    Console.WriteLine($"Error: {error}");
    cancelled = true;
}

while (!cancelled)
{
    consumer.Poll(TimeSpan.FromSeconds(1));
}
```

Для простоты в примере *rd_kafka_commit_message* используется перед обработкой сообщения, так как фиксация каждого сообщения на практике приводит к большим накладным расходам. Поэтому лучшим подходом является сбор пакета сообщений, выполнение синхронного коммита и затем после успешной фиксации обработка сообщений.

Important: Правильное управление смещением имеет решающее значение, поскольку оно влияет на семантику доставки

Асинхронные коммиты

Каждый вызов commit API приводит к отправке брокеру запроса на фиксацию смещения. При использовании синхронного API потребитель блокируется до тех пор, пока запрос не будет успешно возвращен. Это может снизить общую пропускную способность, поскольку в противном случае потребитель мог бы обрабатывать записи, ожидающие фиксации. Одним из способов решения этой проблемы является увеличение объема данных, возвращаемых в каждом *poll()*, через параметр конфигурации *fetch.min.bytes*. Тогда брокер удерживает выборку до тех пор, пока не будет достигнуто достаточное количество данных (или не истечет срок *fetch.max.wait.ms*). Побочный эффект заключается в том, что способ также увеличивает количество дубликатов, с которыми приходится сталкиваться при случае сбоя.

Второй вариант – использовать асинхронные коммиты. Потребитель может отправить запрос и, не дожидаясь завершения запроса, немедленно вернуться.

```
public void run() {
  try {
    consumer.subscribe(topics);

    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      records.forEach(record -> process(record));
      consumer.commitAsync();
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
```

```
    shutdownLatch.countDown();
  }
}
```

C/C++ (*librdkafka*):

```
void consume_loop(rd_kafka_t *rk,
                  rd_kafka_topic_partition_list_t *topics) {
  static const int MIN_COMMIT_COUNT = 1000;

  int msg_count = 0;
  rd_kafka_resp_err_t err;

  if ((err = rd_kafka_subscribe(rk, topics))) {
    fprintf(stderr, "%% Failed to start consuming topics: %s\n", rd_kafka_err2str(err));
    exit(1);
  }

  while (running) {
    rd_kafka_message_t *rkmessage = rd_kafka_consumer_poll(rk, 500);
    if (rkmessage) {
      msg_process(rkmessage);
      rd_kafka_message_destroy(rkmessage);

      if ((++msg_count % MIN_COMMIT_COUNT) == 0)
        rd_kafka_commit(rk, NULL, 1);
    }
  }

  err = rd_kafka_consumer_close(rk);
  if (err)
    fprintf(stderr, "%% Failed to close consumer: %s\n", rd_kafka_err2str(err));
  else
    fprintf(stderr, "%% Consumer closed\n");
}
```

Единственное различие между этим примером и предыдущим заключается в том, что в вызове *rd_kafka_commit* включена асинхронная фиксация.

Изменения в **Python** очень похожи. Параметр *async* для *commit()* изменен на *True*. В примере значение передается явно, но асинхронная фиксация используется по умолчанию, если параметр не включен:

```
def consume_loop(consumer, topics):
    try:
        consumer.subscribe(topics)

        msg_count = 0
        while running:
            msg = consumer.poll(timeout=1.0)
            if msg is None: continue

            if msg.error():
                if msg.error().code() == KafkaError._PARTITION_EOF:
                    # End of partition event
                    sys.stderr.write('%% %s [%d] reached end at offset %d\n' %
                                     (msg.topic(), msg.partition(), msg.offset()))
                elif msg.error():
                    raise KafkaException(msg.error())
```

```
            else:
                msg_process(msg)
                msg_count += 1
                if msg_count % MIN_COMMIT_COUNT == 0:
                    consumer.commit(async=True)
    finally:
        # Close down consumer to commit final offsets.
        consumer.close()
```

В **Go** для асинхронной фиксации необходимо выполнить коммит в goroutine:

```
msg_count := 0
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case *kafka.Message:
        msg_count += 1
        if msg_count % MIN_COMMIT_COUNT == 0 {
            go func() {
                offsets, err := consumer.Commit()
            }()
        }
        fmt.Printf("%% Message on %s:\n%s\n",
            e.TopicPartition, string(e.Value))

    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

В **C#** для асинхронной фиксации необходимо вызвать метод *CommitAsync*:

```
var msgCount = 0;

consumer.OnMessage += (_, msg) =>
{
    processMessage(msg);
    msgCount += 1;
    if (msgCount % MIN_COMMIT_COUNT == 0)
    {
        consumer.CommitAsync();
    }
}

consumer.OnPartitionEOF += (_, end)
    => Console.WriteLine($"Reached end of topic {end.Topic} partition {end.Partition}.");

consumer.OnError += (_, error)
{
    Console.WriteLine($"Error: {error}");
    cancelled = true;
}
```

```
while (!cancelled)
{
    consumer.Poll(TimeSpan.FromSeconds(1));
}
```

Поскольку такой способ помогает производительности, почему бы всегда не использовать асинхронные коммиты? Основная причина заключается в том, что потребитель не повторяет запрос в случае сбоя фиксации. Это то, что *commitSync* предлагает даром; он повторяется бесконечно, пока фиксация не будет выполнена, или не будет найдена неисправимая ошибка. Проблема с асинхронными коммитами связана с порядком фиксации – к тому времени, когда потребитель узнает, что фиксация не удалась, возможно, уже будет обработан следующий пакет сообщений, и даже будет отправлен следующий коммит. В этом случае повторная попытка старой фиксации может привести к дублированию.

Вместо того, чтобы усложнять свойства потребителей в попытках самостоятельного решения этой проблемы, API выдает обратный запрос при свершении коммита – и успешного, и при неудаче. При желании можно использовать этот обратный запрос для повторной фиксации, но тогда приходится сталкиваться с проблемой переназначения.

```
public void run() {
  try {
    consumer.subscribe(topics);

    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      records.forEach(record -> process(record));
      consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception␣
↪exception) {
          if (e != null)
            log.debug("Commit failed for offsets {}", offsets, e);
          }
      });
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
    shutdownLatch.countDown();
  }
}
```

Аналогичная функция доступна в **C/C++** (*librdkafka*), но ее необходимо настроить при инициализации:

```
static void on_commit(rd_kafka_t *rk,
                      rd_kafka_resp_err_t err,
                      rd_kafka_topic_partition_list_t *offsets,
                      void *opaque) {
  if (err)
    fprintf(stderr, "%% Failed to commit offsets: %s\n", rd_kafka_err2str(err));
}

void init_rd_kafka() {
  rd_kafka_conf_t *conf = rd_kafka_conf_new();
  rd_kafka_conf_set_offset_commit_cb(conf, on_commit);
```

```
  // initialization omitted
}
```

Аналогично, в **Python** обратный запрос может быть вызван любым коммитом и может быть передан в качестве параметра конфигурации конструктора потребителя:

```
from confluent_kafka import Consumer

def commit_completed(err, partitions):
    if err:
        print(str(err))
    else:
        print("Committed partition offsets: " + str(partitions))

conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'group.id': "foo",
        'default.topic.config': {'auto.offset.reset': 'smallest'},
        'on_commit': commit_completed}

consumer = Consumer(conf)
```

В **C#** можно использовать *Task*:

```
var msgCount = 0;

consumer.OnMessage += (_, msg) =>
{
    processMessage(msg);
    msgCount += 1;
    if (msgCount % MIN_COMMIT_COUNT == 0)
    {
        consumer.CommitAsync().ContinueWith(
            commitResult =>
            {
                if (commitResult.Error)
                {
                    Console.Error.WriteLine(commitResult.Error);
                }
                else
                {
                    Console.WriteLine(
                        $"Committed Offsets [{string.Join(", ", commitResult.Offsets)}]");
                }
            }
        )
    }
}
```

В **Go** события перебалансировки отображаются как события, возвращаемые методом *Poll()*. Для того чтобы увидеть эти события, необходимо создать потребителя с конфигурацией *go.application.rebalance.enable* и обработать события *AssignedPartitions* и *RevokedPartitions*, явно вызвав *Assign()* и *Unassign()* для *AssignedPartitions* и *RevokedPartitions* соответственно:

```
consumer, err := kafka.NewConsumer(&kafka.ConfigMap{
     "bootstrap.servers":    "host1:9092,host2:9092",
     "group.id":             "foo",
     "go.application.rebalance.enable": true})
```

```
msg_count := 0
for run == true {
    ev := consumer.Poll(0)
    switch e := ev.(type) {
    case kafka.AssignedPartitions:
        fmt.Fprintf(os.Stderr, "%% %v\n", e)
        c.Assign(e.Partitions)
    case kafka.RevokedPartitions:
        fmt.Fprintf(os.Stderr, "%% %v\n", e)
        c.Unassign()
    case *kafka.Message:
        msg_count += 1
        if msg_count % MIN_COMMIT_COUNT == 0 {
            consumer.Commit()
        }

        fmt.Printf("%% Message on %s:\n%s\n",
            e.TopicPartition, string(e.Value))

    case kafka.PartitionEOF:
        fmt.Printf("%% Reached %v\n", e)
    case kafka.Error:
        fmt.Fprintf(os.Stderr, "%% Error: %v\n", e)
        run = false
    default:
        fmt.Printf("Ignored %v\n", e)
    }
}
```

Сбои фиксации смещения досаждают, когда последующие коммиты успешны, так как фактически они не должны приводить к повторным чтениям. Однако, если последняя фиксация завершается неудачно до того, как происходит перебалансировка или отключение потребителя, смещения сбрасываются до последнего коммита, и вероятнее всего, отображаются дубликаты. Поэтому общая схема заключается в том, чтобы объединить асинхронные коммиты в цикле опроса с синхронизированными коммитами при перебалансировках или отключении. Фиксация при отключении несложна, но необходимо найти способ закрепления поведения при перебалансировке. Для этого представленный ранее метод *subscribe()* имеет вариант, принимающий *ConsumerRebalanceListener*, который имеет два метода закрепления поведения перебалансировки.

В следующем примере синхронные фиксации включаются при перебалансировках и при отключении:

```
private void doCommitSync() {
  try {
    consumer.commitSync();
  } catch (WakeupException e) {
    // we're shutting down, but finish the commit first and then
    // rethrow the exception so that the main loop can exit
    doCommitSync();
    throw e;
  } catch (CommitFailedException e) {
    // the commit failed with an unrecoverable error. if there is any
    // internal state which depended on the commit, you can clean it
    // up here. otherwise it's reasonable to ignore the error and go on
    log.debug("Commit failed", e);
  }
}

public void run() {
  try {
```

```
    consumer.subscribe(topics, new ConsumerRebalanceListener() {
      @Override
      public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        doCommitSync();
      }

      @Override
      public void onPartitionsAssigned(Collection<TopicPartition> partitions) {}
    });

    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(Long.MAX_VALUE);
      records.forEach(record -> process(record));
      consumer.commitAsync();
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    try {
      doCommitSync();
    } finally {
      consumer.close();
      shutdownLatch.countDown();
    }
  }
}
```

Каждая перебалансировка имеет две фазы: отзыв и назначение партиции. Отзыв партиции всегда вызывается перед перебалансировкой и является последним шансом фиксации смещения перед переназначением. Фаза назначения партиции всегда вызывается после перебалансировки и может использоваться для установки начальной позиции назначенных партиций. В этом случае отзыв используется для синхронной фиксации текущих смещений.

Как правило, асинхронные коммиты следует считать менее безопасными, чем синхронные, так как последовательные неудачи фиксации приводят к увеличению обработки дубликатов. Можно снизить эту опасность, добавив логику для обработки ошибок фиксации в обратном запросе или периодически смешивая с вызовами *commitSync()*, но не следует добавлять слишком много сложностей при отсутствии прямой необходимости. При синхронных коммитах можно повысить надежность, увеличив число партиций топика и количество потребителей в группе. А если необходимо максимизировать пропускную способность при готовности некоторого увеличения числа дубликатов, то асинхронные коммиты могут стать хорошим вариантом.

Довольно очевидный момент, но стоит отметить, что асинхронные коммиты имеют смысл только для “at least once” доставки сообщений. Чтобы получить “at most once”, прежде чем считывать сообщение необходимо знать, успешна ли фиксация. А это подразумевает синхронную фиксацию, за исключением случая наличия возможности “непрочтения” сообщения после того, как обнаружится, что фиксация не удалась.

Мониторинг групп

Kafka включает в себя утилиту администратора для просмотра статуса групп потребителей.

Для получения списка активных групп в кластере можно использовать утилиту **kafka-consumer-groups**, включенную в дистрибутив **Kafka**. В большом кластере это может занять некоторое время, поскольку она собирает список путем проверки каждого брокера.

```
bin/kafka-consumer-groups --bootstrap-server host:9092 --list
```

Утилита **kafka-consumer-groups** также может быть использована для сбора информации о текущей группе. Пример просмотра актуальных назначений для группы *foo*:

```
bin/kafka-consumer-groups --bootstrap-server host:9092 --describe --group foo
```

В случае вызова утилиты во время перебалансировки, команда сообщает об ошибке. Тогда необходимо повторить попытку, в результате которой отображаются назначения для всех членов в текущей группе.

2.3 Kafka Java Producer

Платформа **ADS** включает в себя Java producer, поставляемый вместе с **Kafka**.

В документе представлен общий обзор работы поставщика, введение в параметры конфигурации для настройки и примеры из каждой клиентской библиотеки.

2.3.1 Концепция

Поставщик **Kafka** концептуально намного проще, чем потребитель, так как у него нет необходимости в групповой координации. Его основная функция состоит в том, чтобы сопоставить каждое сообщение с партицией топика и отправить запрос лидеру соответствующей партиции. Первое выполняется с помощью partitioner – механизма, выбирающего партицию посредством хэш-функции, и гарантирующего, что все сообщения с одинаковым (непустым) ключом отправляются в одну и ту же партицию. Если ключ не указан, то партиция определяется циклически с целью обеспечения равномерного распределения по партициям топика.

В каждой партиции кластера **Kafka** есть лидер и набор реплик среди брокеров – все записи проходят через лидера партиции, а реплики синхронизируются посредством выборки из него. Когда лидер отключается или выходит из строя, следующий лидер выбирается из числа синхронизированных реплик. В зависимости от того, как настроен поставщик, каждый запрос лидеру партиции может удерживаться до тех пор, пока реплики не одобрят запись. Это дает поставщику некий контроль над эксплуатацией сообщения при условии некоторой стоимости общей пропускной способности.

Сообщения, направленные лидеру партиции, не могут сразу считываться потребителями независимо от настроек подтверждения поставщика. Только когда все синхронизированные реплики подтверждают запись, сообщение считается зафиксированным, что делает его доступным для чтения. Такой подход гарантирует, что уже прочитанные сообщения не могут быть потеряны по причине сбоя брокера. Но это также подразумевает, что сообщения, подтвержденые только лидером (то есть *acks=1*), могут быть потеряны в случае, если лидер партиции терпит неудачу до момента копирования сообщений репликами. Тем не менее, в большинстве случаев на практике такой способ часто является разумным компромиссом для обеспечения жизнеспособности сообщений (durability), при этом не оказывая существенного влияния на пропускную способность.

Большая часть тонкостей вокруг поставщиков связана с достижением высокой пропускной способности с учетом пакетирования/сжатия и обеспечением гарантий доставки сообщений. Далее приведены наиболее распространенные параметры настройки поведения поставщиков.

2.3.2 Конфигурация

Полный список параметров конфигурации доступен в документе Настройки платформы Arenadata Streaming. Но некоторые из ключевых параметров и их влияние на поведение поставщиков описаны в текущей главе.

Базовая конфигурация (Core Configuration)

Для того, чтобы поставщик мог найти кластер **Kafka**, необходимо установить свойство *bootstrap.servers*. Так же, хотя это и не требуется обязательно, но всегда следует устанавливать *client.id*, поскольку это позволяет легко сопоставлять запросы в брокере со сделавшим их инстансом клиента. Данные настройки одинаковы для клиентов **Java**, **C/C++**, **Python**, **Go** и **.NET**.

Жизнеспособность сообщений (Message Durability)

Жизнеспособность сообщений, записанных в **Kafka**, можно контролировать с помощью параметра *acks*. Значение по умолчанию *1* требует от лидера партиции явного подтверждения об успешно выполненной записи. Самая сильная гарантия, которую предоставляет **Kafka** – *acks=all* – сообщение не только допущено к записи лидером партиции, но и успешно скопировано на все синхронизированные реплики. Можно также использовать значение *0* для максимизации пропускной способности, но тогда отсутствует гарантия успешной записи сообщения в журнал брокера, так как в этом случае брокер не отправляет ответ, что также означает невозможность определения смещение сообщения. Для клиентов **C/C++**, **Python**, **Go** и **.NET** это является конфигурацией для каждого отдельного топика, но ее можно применять глобально с помощью вложенной конфигурации *default_topic_conf* в **C/C++** и *default.topic.config* в **Python**, **Go** и **.NET**.

Порядок сообщений (Message Ordering)

Как правило, сообщения записываются в брокер в том же порядке, в котором они поступают от клиента поставщика. Однако если разрешить повторные попытки сообщений, установив для них значение больше *0* (*0* – значение по умолчанию), может измениться их порядок, так как повтор возможен только после свершения успешной записи. Чтобы избежать переупорядочения, можно установить параметр *max.in.flight.requests.per.connection* в значение *1*, тогда брокеру одновременно может быть отправлен только один запрос. В случае без подключения повторных попыток сообщений брокер сохраняет порядок получаемых записей, но при этом могут быть пробелы из-за отдельных сбоев отправки.

Пакетирование и сжатие (Batching/Compression)

Поставщики **Kafka** пакетируют отправляемые сообщения для повышения пропускной способности. С клиентом **Java** можно управлять максимальным размером каждого пакета сообщений в параметре *batch.size*. Для большего времени на заполнение пакетов доступен параметр *linger.ms*, на значение которого поставщик задерживает отправку. Установкой *compression.type* включается сжатие, оно охватывает полные пакеты сообщений, поэтому большие пакеты обычно означают более высокую степень сжатия.

С клиентами **C/C++**, **Python**, **Go** и **.NET** для установки ограничения на количество сообщений в пакете используется *batch.num.messages*, сжатие включается с помощью *compression.codec*.

Ограничения очереди (Queuing Limits)

Ограничение общего объема доступной Java-клиенту памяти для сбора неотправленных сообщений контролируется параметром *buffer.memory*. При достижении установленного предела поставщик блокирует последующий набор записей до тех пор, пока *max.block.ms* не выводит исключение. Кроме того, чтобы избежать бесконечного хранения сообщений, можно установить тайм-аут в *request.timeout.ms*. Если это время ожидания истекает до того, как сообщение может быть успешно отправлено, то оно удаляется из очереди и генерируется исключение.

Клиенты **C/C++**, **Python**, **Go** и **.NET** имеют аналогичные настройки. Параметр *queue.buffering.max.messages* – для ограничения общего количества сообщений, поставляемых в очередь в любой момент времени (для отчетов о передаче, повторных попытках или доставке). И параметр *queue.buffering.max.ms* – для ограничения периода времени ожидания клиентом заполнения пакета перед отправкой брокеру.

2.3.3 Примеры

Начальная настройка

Поставщик **Java** создается с помощью стандартного файла свойств *Properties*:

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("bootstrap.servers", "host1:9092,host2:9092");
```

```
config.put("acks", "all");
new KafkProducer<K, V>(config);
```

Ошибки конфигурации приводят к появлению *KafkaException* от конструктора *KafkaProducer*. Основное отличие *librdkafka* заключается в том, что она обрабатывает ошибки для каждого параметра напрямую:

```
char hostname[128];
char errstr[512];

rd_kafka_conf_t *conf = rd_kafka_conf_new();

if (gethostname(hostname, sizeof(hostname))) {
  fprintf(stderr, "%% Failed to lookup hostname\n");
  exit(1);
}

if (rd_kafka_conf_set(conf, "client.id", hostname,
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
  fprintf(stderr, "%% %s\n", errstr);
  exit(1);
}

if (rd_kafka_conf_set(conf, "bootstrap.servers", "host1:9092,host2:9092",
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
  fprintf(stderr, "%% %s\n", errstr);
  exit(1);
}

rd_kafka_topic_conf_t *topic_conf = rd_kafka_topic_conf_new();

if (rd_kafka_topic_conf_set(topic_conf, "acks", "all",
                      errstr, sizeof(errstr)) != RD_KAFKA_CONF_OK) {
  fprintf(stderr, "%% %s\n", errstr);
  exit(1);
}

/* Create Kafka producer handle */
rd_kafka_t *rk;
if (!(rk = rd_kafka_new(RD_KAFKA_PRODUCER, conf,
                        errstr, sizeof(errstr)))) {
  fprintf(stderr, "%% Failed to create new producer: %s\n", errstr);
  exit(1);
}
```

В **Python**:

```
from confluent_kafka import Producer
import socket

conf = {'bootstrap.servers': "host1:9092,host2:9092",
        'client.id': socket.gethostname(),
        'default.topic.config': {'acks': 'all'}}

producer = Producer(conf)
```

В **Go**:

```
import (
    "github.com/confluentinc/confluent-kafka-go/kafka"
)

p, err := kafka.NewProducer(&kafka.ConfigMap{
    "bootstrap.servers": "host1:9092,host2:9092",
    "client.id": socket.gethostname(),
    "default.topic.config": kafka.ConfigMap{'acks': 'all'}
})

if err != nil {
    fmt.Printf("Failed to create producer: %s\n", err)
    os.Exit(1)
}
```

В **C#**:

```
using Confluent.Kafka;
using System.Net;

...

var config = new Dictionary<string, object>
{
    { "bootstrap.servers", "host1:9092,host2:9092" },
    { "client.id", Dns.GetHostName() },
    { "default.topic.config", new Dictionary<string, object>
        {
            { "acks", "all" }
        }
    }
}

using (var producer = new Producer<Null, string>(config, null, new StringSerializer(Encoding.
↪UTF8)))
{
    ...
}
```

Асинхронные записи

Все записи являются асинхронными по умолчанию. Поставщик **Java** включает в себя API *send()*, возвращающий future, которое можно опрашивать для получения результата отправки:

```
final ProducerRecord<K, V> = new ProducerRecord<>(topic, key, value);
Future<RecordMetadata> future = producer.send(record);
```

В *librdkafka* сначала необходимо создать дескриптор *rd_kafka_topic_t* для топика, в который планируется запись, а затем использовать *rd_kafka_produce* для отправки в него сообщений. Например:

```
rd_kafka_topic_t *rkt = rd_kafka_topic_new(rk, topic, topic_conf);

if (rd_kafka_produce(rkt, RD_KAFKA_PARTITION_UA,
                     RD_KAFKA_MSG_F_COPY,
                     payload, payload_len,
                     key, key_len,
                     NULL) == -1) {
```

```
  fprintf(stderr, "%% Failed to produce to topic %s: %s\n",
          topic, rd_kafka_err2str(rd_kafka_errno2err(errno)));
}
```

Конкретную топику конфигурацию можно назначить третьему аргументу *rd_kafka_topic_new* – тогда необходимо передать *topic_conf* и добавить настройку для подтверждений. Значение *NULL* приводит к использованию поставщиком конфигурации по умолчанию.

Второй аргумент для *rd_kafka_produce* может использоваться для установки желаемой партиции для сообщения. При установленом значении *RD_KAFKA_PARTITION_UA*, как в данном примере, выбор партиции для сообщения осуществляется механизмом partitioner по умолчанию. Третий аргумент указывает, что *librdkafka* должна скопировать информацию и ключ, что позволяет освободить его по возвращении.

В **Python** отправка инициируется методом *produce* с передачей значения и по необходимости – ключа, партиции и обратного вызова. Запрос возвращается немедленно без значения:

```
producer.produce(topic, key="key", value="value")
```

Аналогично, в **Go** отправка инициируется методом *Produce()* с передачей объекта *Message‘ object and an optional ‘‘chan Event*, применяемого для прослушивания результата отправки. Объект *Message* содержит непрозрачное поле *interface{}*, которое может использоваться для передачи произвольных данных вместе с сообщением последующему обработчику событий.

```
delivery_chan := make(chan kafka.Event, 10000)
err = p.Produce(&kafka.Message{
    TopicPartition: kafka.TopicPartition{Topic: "topic", Partition: kafka.PartitionAny},
    Value: []byte(value)},
    delivery_chan,
)
```

В **C#** отправка инициируется вызовом метода *ProduceAsync* в инстансе *Producer*. Например:

```
producer.ProduceAsync("topic", key, value);
```

При необходимости применения некоторого кода после завершения записи может быть предоставлен обратный вызов. В **Java** это реализовано как объект *Callback*:

```
final ProducerRecord<K, V> = new ProducerRecord<>(topic, key, value);
producer.send(record, new Callback() {
  public void onCompletion(RecordMetadata metadata, Exception e) {
    if (e != null)
      log.debug("Send failed for record {}", record, e);
  }
});
```

В реализации **Java** следует избегать дорогостоящей работы с обратным вызовом, поскольку он выполняется в потоке ввода-вывода поставщика.

Аналогичная функция доступна в *librdkafka*, но ее необходимо настраивать при инициализации:

```
static void on_delivery(rd_kafka_t *rk,
                        const rd_kafka_message_t *rkmessage
                        void *opaque) {
  if (rkmessage->err)
    fprintf(stderr, "%% Message delivery failed: %s\n",
            rd_kafka_message_errstr(rkmessage));
}
```

```
void init_rd_kafka() {
  rd_kafka_conf_t *conf = rd_kafka_conf_new();
  rd_kafka_conf_set_dr_msg_cb(conf, on_delivery);

  // initialization ommitted
}
```

Обратный вызов доставки (delivery callback) в *librdkafka* осуществляется в потоке пользователя путем вызова *rd_kafka_poll*. Распространенным шаблоном является вызов функции после каждого вызова API поставщика, но этого может быть недостаточно для обеспечения регулярных отчетов о доставке, если скорость создания сообщений не равномерна. Так же данный API не предоставляет прямого способа блокировки для получения результата доставки конкретного сообщения. При наличии такой необходимости рекомендуется рассмотреть пример синхронной записи (*Синхронные записи*).

В **Python** параметр *callback* можно передать с помощью любого вызываемого средства, например, лямбды, функции, связанного метода или вызываемого объекта. Хотя метод *produce()* сразу ставит сообщение в очередь для пакетной обработки, сжатия и передачи брокеру, он не свершает обработку каких-либо событий (то есть подтверждений и обратных вызовов, которые они инициируют) до вызова *poll()*.

```
def acked(err, msg):
    if err is not None:
        print("Failed to deliver message: %s: %s" % (str(msg), str(err))
    else
        print("Message produced: %s" % (str(msg))

producer.produce(topic, key="key", value="value", callback=acked)

# Wait up to 1 second for events. Callbacks will be invoked during
# this method call if the message is acknowledged.
producer.poll(1)
```

В **Go** можно использовать канал отчета о доставке в *Produce*, чтобы дождаться результата отправки сообщения:

```
e := <-delivery_chan
m := e.(*kafka.Message)

if m.TopicPartition.Error != nil {
    fmt.Printf("Delivery failed: %v\n", m.TopicPartition.Error)
} else {
    fmt.Printf("Delivered message to topic %s [%d] at offset %v\n",
               *m.TopicPartition.Topic, m.TopicPartition.Partition, m.TopicPartition.Offset)
}

close(delivery_chan)
```

В **C#** есть два варианта. Первый: можно использовать *ProduceAsync*, возвращающий стандартный объект *Task*, который выполняет await (приостановку выполнения метода до завершения выполнения ожидаемой задачи), обрабатывается с помощью метода *.ContinueWith* или ожидает использования методов *.Wait* или *.WaitAll*:

```
var deliveryReportTask = producer.ProduceAsync("topic", key, val);
deliveryReportTask.ContinueWith(task =>
{
    Console.WriteLine($"Partition: {task.Result.Partition}, Offset: {task.Result.Offset}");
});
```

Во втором варианте используется *.ProduceAsync*, который принимает реализацию *IDeliveryHandler*. Данный подход следует использовать при необходимости получения уведомлений о доставке сообщений (или сбое доставки) строго в порядке подтверждения брокером, поскольку *Tasks* могут выполняться в любом потоке, что не гарантирует их упорядоченность.

Синхронные записи

Чтобы сделать запись синхронной, следует дождаться возвращения future. Как правило, это плохая затея, так как она может существенно снизить пропускную способность, но в некоторых случаях может быть оправдана.

```
Future<RecordMetadata> future = producer.send(record);
RecordMetadata metadata = future.get();
```

Аналогичная возможность может быть достигнута в **C/C++** и **Python** с помощью обратного вызова доставки, но это более трудоемко. Полный пример приведен по ссылке. Клиент **Python** также содержит метод *flush()*, имеющий тот же эффект:

```
producer.produce(topic, key="key", value="value")
producer.flush()
```

В **Go** осуществляется через канал доставки, посредством вызова метода *Produce()*:

```
delivery_chan := make(chan kafka.Event, 10000)
err = p.Produce(&kafka.Message{
    TopicPartition: kafka.TopicPartition{Topic: "topic", Partition: kafka.PartitionAny},
    Value: []byte(value)},
    delivery_chan
)

 e := <-delivery_chan
 m := e.(*kafka.Message)
```

Для ожидания подтверждения всех сообщений используется метод *Flush()*:

```
p.Flush()
```

Важно обратить внимание, что *Flush()* обслуживает только канал *Events()* поставщика, а не каналы доставки, указанные приложением. Если *Flush()* вызывается, и при этом никакая горутина не обрабатывает канал доставки, то буфер может заполниться и привести к истечению времени ожидания.

В **C#** необходимо получить доступ к свойству *.Result* объекта *Task*, возвращенного из *.ProduceAsync*, которое будет блокироваться до тех пор, пока не станет доступен отчет о доставке:

```
var deliveryReport = producer.ProduceAsync("topic", key, value).Result;
```

2.4 Kafka Connect

Kafka Connect – компонент **Apache Kafka** с открытым исходным кодом, является основой для подключения **Kafka** к внешним системам, таким как базы данных, хранилища key-value, поисковые индексы и файловые системы. С **Kafka Connect** можно использовать существующие реализации коннекторов для перемещения данных в сервис **Kafka** и из него:

- *Source Connector* – принимает базы данных и обновляет таблицы потоков для топиков **Kafka**. Также собирает метрики со всех серверов приложений в топики **Kafka**, делая данные доступными для потоковой обработки с низкой задержкой;

- *Sink Connector* – доставляет данные из топиков **Kafka** во вторичные индексы, такие как **Elasticsearch**, или в пакетные системы для автономного анализа, такие как **Hadoop**.

Kafka Connect ориентирован на потоковую передачу данных из сервиса **Kafka** и в него, что упрощает написание высококачественных, надежных и высокопроизводительных плагинов. Это также позволяет фреймворку давать гарантии, которые трудно достичь с помощью других структур. **Kafka Connect** является неотъемлемым компонентом конвейера **ETL** в сочетании с сервисом **Kafka** и потоковой обработкой.

Kafka Connect может работать либо как автономный процесс для выполнения заданий на одной машине (например, сбор журналов), либо как распределенный, масштабируемый, отказоустойчивый сервис, поддерживающий всю структуру. Это позволяет сократить масштаб до разработки, тестирования и небольших продуктовых развертываний с низким барьером для входа и низкими эксплуатационными накладными расходами, а также увеличить масштаб поддержки конвейера данных большой организации.

Основные преимущества использования **Kafka Connect**:

- *Data Centric Pipeline* – использование значимых абстракций данных для извлечения или передачи данных в **Kafka**;
- *Flexibility and Scalability (гибкость и масштабируемость)* – работа с потоковыми и пакетно-ориентированными системами на одном узле или масштабирование до сервиса по всей ширине организации;
- *Reusability and Extensibility (повторное использование и расширяемость)* – использование существующих коннекторов и возможность расширения их для адаптации к конкретным потребностям и сокращения времени на разработку.

2.4.1 Connectors & Tasks

Копирование данных между сервисом **Kafka** и сторонней системой осуществляется посредством создаваемых пользователями инстансов *Kafka Connectors*. Коннекторы бывают двух видов: *SourceConnectors* – импортируют данные из другой системы, и *SinkConnectors* – экспортируют данные в другую систему. Например, *JDBCSourceConnector* импортирует реляционную базу данных в **Kafka**, а *HDFSSinkConnector* экспортирует содержимое топика **Kafka** в файлы **HDFS**.

Реализации класса Connector не выполняют копирование данных самостоятельно: их конфигурация описывает набор данных для копирования, и Connector отвечает за разбиение этого задания на набор задач – Tasks, которые могут быть распределены между объектами **Kafka Connect**. Tasks также бывают двух видов: SourceTask и SinkTask. При необходимости реализация класса Connector может отслеживать изменения данных внешних систем и запрашивать реконфигурацию задачи.

С назначением данных, которые должны быть скопированы, каждая задача *Task* должна скопировать свое подмножество данных в сервис **Kafka** или из него. Данные, которые копирует коннектор, должны быть представлены как партиционированный поток, аналогично модели топика **Kafka**, где каждая партиция представляет собой упорядоченную последовательность записей со смещениями. Каждой задаче назначается подмножество партиций для обработки. Порой это сопоставление очевидно: каждый файл в наборе файлов журнала можно считать партицией, каждую строку в файле – записью, а смещения – просто позициии в файле. В иных случаях сопоставление с моделью требует больше усилий: коннектор **JDBC** может сопоставить каждую таблицу с партицией, но смещение менее ясно. Один из возможных вариантов сопоставления это использовать в качестве смещения последнюю запрашиваемую отметку времени при генерации запросов.

Source Connector, создавший две задачи, которые копируют данные из входных партиций и записывают в сервис **Kafka**, приведен на Рис.2.1..

2.4.2 Partitions & Records

Каждая партиция представляет собой упорядоченную последовательность записей ключ-значение, где и ключи, и значения могут иметь сложные структуры. Поддерживаются многие примитивные типы, а также массивы, структуры и вложенные структуры данных. Для большинства типов можно напрямую использовать

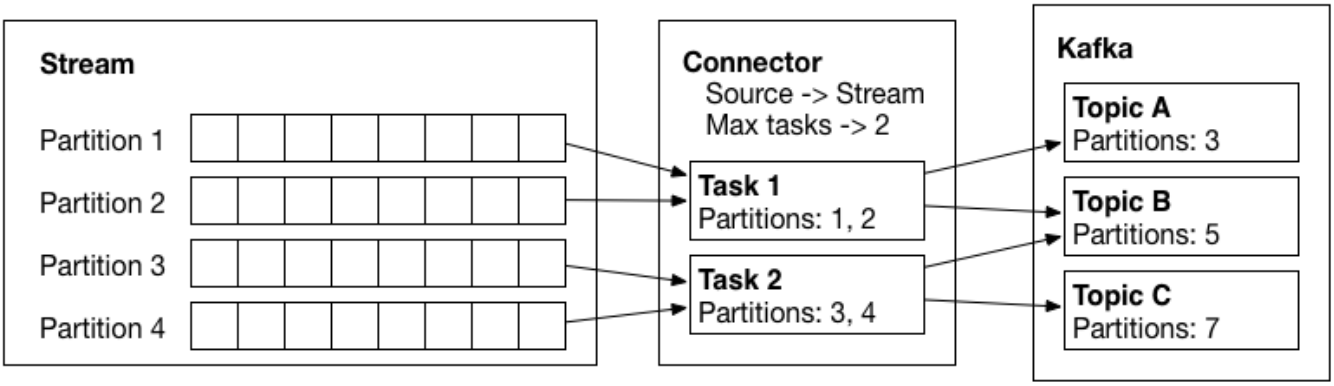

Рис.2.1.: Пример реализации *Source Connector*

стандартные типы **Java**, такие как *java.lang.Integer*, *java.lang.Map* и *java.lang.Collection*. Для структурированных записей следует использовать класс *Struct*.

На Рис.2.2. представлен партиционированный поток: модель данных, в которой коннекторы сопоставляют все системы *source* и *sink*. Каждая запись содержит ключи и значения (со схемами), идентификатор партиции и смещения в ней.

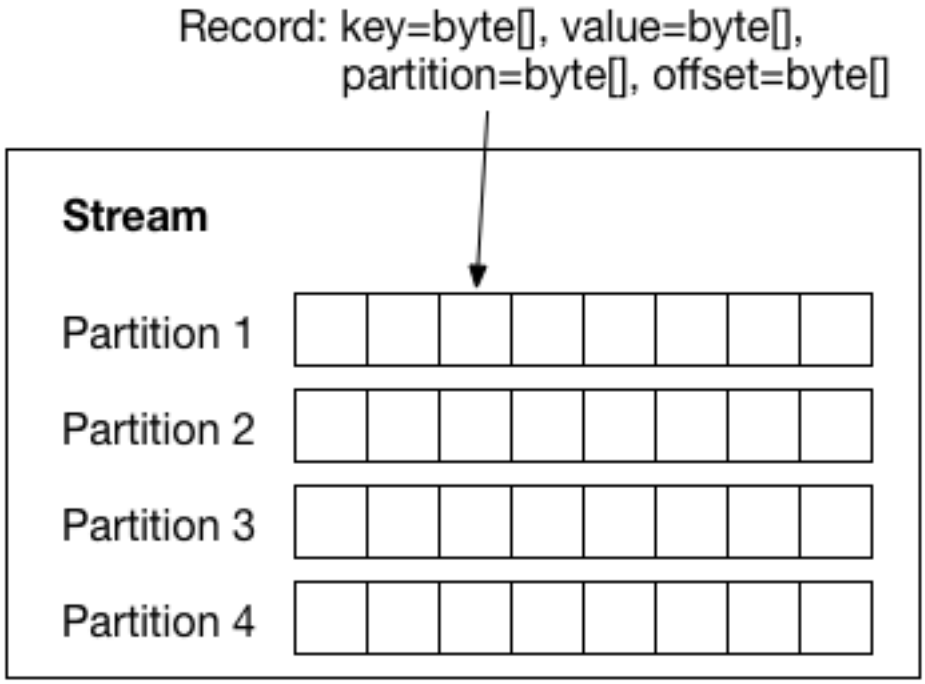

Рис.2.2.: Пример партиционированного потока

Для отслеживания структуры и совместимости записей в партициях схемы (*Schemas*) могут быть включены в каждую запись. Поскольку схемы обычно генерируются “на лету” на основе источника данных, класс *SchemaBuilder* включен, что делает их построение очень простым.

Schemas Определение абстрактного типа данных. Типы данных могут быть примитивными типами

(целочисленные типы, типы с плавающей запятой, логические, строки и байты) или сложными типами (типизированные массивы, карты с одной схемой ключей и схемами значений, а также структурами, которые имеют фиксированный набор имен полей, каждый из которых имеет схема связанных значений). Любой тип может быть указан как необязательный, что позволяет его опускать (в результате чего значения отсутствуют) и может указывать значение по умолчанию.

Такой формат данных среды выполнения не предполагает какого-либо конкретного формата сериализации; это преобразование осуществляется с помощью *Converter*, которые обрабатывают формат времени выполнения *org.apache.kafka.connect.data* и сериализованные данные *byte[]*.

Converter Интерфейс конвертера обеспечивает поддержку перевода между форматом данных выполнения Kafka Connect и byte[]. Внутренне это включает промежуточный шаг к формату, используемому слоем сериализации (например, JsonNode, GenericRecord, Message).

В дополнение к ключу и значению записи имеют идентификаторы партиций и смещения, которые используются фреймворком для периодической фиксации смещений обработанных данных. В случае сбоя обработка может возобновиться с последнего зафиксированного смещения, что позволяет избежать повторной обработки и дублирования событий.