Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Балакина Екатерина Сергеевна

Реализация межпроцессного взаимодействия на контроллере ТРИК

Курсовая работа

Научный руководитель:
ст. преп. Я. А. Кириленко

Санкт-Петербург
2018

Оглавление

Введение **3**

1. Постановка задачи **5**

2. Требования к решению **6**

3. Обзор **8**

3.1. Разделяемая память 8
3.2. Очередь сообщений 8
3.3. Семафоры 9
3.4. Каналы 9
3.5. Unix Domain Sockets 11
3.6. D-Bus 12
3.7. RabbitMQ 14
3.8. ZeroMQ 15
3.9. nanomsg 16

4. Тестирование **17**

5. Архитектура **20**

6. Заключение **22**

Список литературы **23**

Введение

В современном мире роботы становятся все более востребованы. Их активно используют в медицине и строительстве отправляют в космос, им поручают следить за безопасностью людей и их жилищ. Роботы незаменимы в промышленности и быту, справляясь со многими повседневными задачами быстрее и качественнее людей. Повсеместная распространенность роботов привела к возникновению целой науки — робототехники, в числе направлений которой также находится разработка инструментария, упрощающего создание и программирование роботов.

Одним из примеров подобного инструментария является ТРИК [8] — кибернетический контроллер и одноименный конструктор, которые вместе предоставляют платформу для создания роботов. Контроллер ТРИК обладает вычислительными ресурсами, достаточными для решения нетривиальных робототехнических задач и реализации сложных алгоритмов, работающих в реальном времени с постоянно изменяющимся потоком входных данных.

В ТРИК существует понятие абстрактного датчика — это программный код, который передает в управляющую программу некоторые данные, являющиеся результатом некоторой обработки информации (например, аудио- или видеоряд) с периферийных устройств, или же данные от внешних сервисов наподобие Twitter. Эти обработанные данные передаются пользовательским программам, которые используют их для своей работы. Программные датчики и программы, использующие информацию с этих датчиков, могут быть реализованы с помощью различных языков программирования и технологий, поэтому они работают в разных процессах.

Между тем на данный момент нет единого интерфейса обмена данными между различными процессами на контроллере, что затрудняет их использование.

В рамках данной работы представлены исследования по реализации средства взаимодействия между процессами на платформе ТРИК.

1. Постановка задачи

Цель представляемой работы — разработать интерфейс, который позволит процессам различной природы и происхождения на контроллере оперативно обмениваться информацией. В рамках данной курсовой работы были поставлены следующие задачи:

1. проанализировать современные средства межпроцессного взаимодействия и готовые решения, использующие их;
2. разработать архитектуру новой подсистемы интеграции процессов.

2. Требования к решению

Прежде чем переходить к описанию требований, которым должно удовлетворять решение задачи, стоит чуть подробнее описать модель взаимодействия программных сенсоров на контроллере с другими процессам.

Центральный ARM процессор контроллера ТРИК работает под управлением операционной системы Linux. Среда исполнения trikRuntime содержит библиотеку trikControl, обеспечивающую объектно-ориентированный доступ к сенсорам, прослойку для конфигурации и некоторую логику обработки полученной информации. Для процессов, написанных на других языках программирования, для доступа к сенсорам есть несколько вариантов.

1. Обращаться к ним через модуль trikControl, написанном на языке программирования C++;

2. Обращаться напрямую к процессу, представляющему сенсор. В таком случае всю высокоуровневую логику обработки данных придется дублировать. В некоторых случаях разделение доступа к ресурсам, если они уже «подцеплены» из trikControl, затруднительно. Например, старый способ подключения геймпад через TCP делает геймпад недоступным для других процессов, а новый с использованием FIFO – позволяет подключиться другим процессам, но не решает проблему совместного доступа.

С учетом существующей модели и задач, для решения которых разрабатывается новая модель межпроцессного взаимодействия, к решению были предъявлены следующие требования, перечисленные ниже.

1. Связь «один ко многим». Это и оговоренный выше доступ из разных языков программирования, и распределение информации между всеми нуждающимися в ней процессами (broadcast).

2. Двухсторонняя связь. На некоторых программных сенсорах на контроллере есть возможность их настройки через стандартный ввод.

3. Возможность запуска или приостановки сенсора по требованию.

4. Трафик — приблизительно 80 сообщений в секунду по 8-10 байт.

5. Доступ к выбранному IPC из разных языков программирования.

3. Обзор

Для разработки новой модели взаимодействия процессов на контроллере с учетом требований, описанных выше, был проведен обзор некоторых существующих низкоуровневых механизмов межпроцессного взаимодействия (Inter-process communication — IPC) и систем на их основе.

В POSIX-совместимых операционных системах, наряду с именованными каналами, чаще всего используются следующие механизмы IPC: разделяемая память, очередь сообщений, семафоры и каналы.

3.1. Разделяемая память

Представляет собой сегмент памяти, совместно используемый несколькими процессами («отображенное в память IPC»). Обмен информации происходит без использования системных вызовов ядра, что делает этот механизм самым быстрым.

3.2. Очередь сообщений

Обеспечивают асинхронный обмен данными между процессами. Процесс-источник посредством некоторых системных вызовов помещает сообщение в очередь, а любой другой процесс может прочитать его оттуда, при условии, что и процесс-источник сообщения и процесс-приемник сообщения используют один и тот же ключ для получения доступа к очереди. Стоит сказать, что очередь сообщений — это, вообще говоря, парадигма сродни паттерну издатель/подписчик, имеющая не только низкоуровневую POSIX реализацию, но и множество высокоуровневых. Существует несколько стандартов, исполь-

зующихся в реализации механизма очередей, в частности, AMQP (Advanced Message Queuing Protocol), легший в основу платформы RabbitMQ.

3.3. Семафоры

Механизм, использующийся для синхронизации потоков и процессов при обращении к разделяемой памяти и критическим участкам кода. При работе с семафорами доступны 3 операции: инициализация (init), захват (wait), освобождение (post). Существует два вида семафоров: именованные и безымянные. Первые могут быть использованы для синхронизации между несколькими несвязанными процессами. Вторые – потоками внутри одного процесса или для синхронизации между родственными процессами.

Эти три механизма: семафоры, разделяемая память и очередь сообщений — являются средствами IPC, соответствующими стандарту POSIX. Это низкоуровневые механизмы, не обладающие никакой «умной» логикой. Можно, конечно, строить свое решение на их основе, но в этом нет необходимости, учитывая, что существуют высокоуровневые, более удобные средства, вполне удовлетворяющие требованиям задачи.

3.4. Каналы

Неименованные каналы – одно из первых средств IPC, появившиеся для UNIX. Они обладают существенным недостатком: не имеют имени и поэтому могут использоваться только родственными процессами. Именованные каналы (FIFO) функционируют подобно обыч-

ным, но существуют в виде специального файла устройства в файловой системе и обладают именем, что позволяет двум несвязанным процессам разделять данные через такой канал. В отличие от традиционного «безымянного» канала, именованный канал остается в файловой системе для дальнейшего использования и после того, как весь ввод/вывод сделан.

Одним из главных преимуществ использования FIFO является то, что именованный канал позволяет различным процессам обмениваться данными, даже если программы, выполняющиеся в этих процессах, изначально не были написаны для взаимодействия друг с другом.

На момент написания работы на контроллере уже была реализована поддержка именованных каналов. Этот механизм не предъявляет практически никаких требований к клиенту и серверу — он в принципе не разделяет клиент и сервер — достаточно лишь того, чтобы оба конца канала умели взаимодействовать со стандартным потоком ввода и вывода. Однако, вероятно, FIFO не самый оптимальный выбор для интеграции процессов. Он неплохо подходит для организации взаимодействия один к одному, но для организации взаимодействия один ко многим требуются некоторые специальные действия. FIFO не является разделяемым ресурсом - несколько процессов могут получить доступ на чтение, но одна и та же информация не может быть прочитана дважды. (O в аббревиатуре FIFO означает Out, данные при чтении извлекаются из файла). Канал FIFO не является двусторонним – для связи между процессами в обе стороны требуется 2 канала. Определенные проблемы возникают из-за блокирования – следует особенно аккуратно определять порядок, в котором FIFO открываются на чтение и запись, а также использовать флаг

O_NONBLOCK, чтобы отменить действие блокирования.

3.5. Unix Domain Sockets

Для организации межпроцессного взаимодействия в пределах одной машины используются UNIX domain сокеты [4]. Сокеты представляют собой виртуальный объект, который существует, пока на него ссылается хотя бы один из процессов. В отличие от TCP/IP сокетов, обращение к локальному сокету происходит по имени файлового пути. Так же, как и в случае с именованными каналами, UNIX Domain сокеты представляются процессами как индексные дескрипторы в файловой системе. Это позволяет двум различным процессам открывать один и тот же сокет для взаимодействия между собой. Однако конкретное взаимодействие, обмен данными использует не файловую систему, а только буферы памяти ядра. В отличие от именованных каналов, при использовании сокетов явно разделяются клиент и сервер, а интерфейс сокета может обслуживать несколько клиентов в двух направлениях. Сокеты являются стандартной компонентой почти всех современных операционных систем, для их использования не требуется устанавливать никаких дополнительных библиотек. Подобно TCP-сокетам, UNIX domain поддерживают потоковую передачу и гарантирует целостность полученных данных. Также сокеты могут работать в режимах передачи датаграмм: упорядоченной и надежной передачи или неупорядоченной и ненадежной (как в UDP-протоколе).

3.6. D-Bus

D-Bus [3] — механизм межпроцессного взаимодействия и удаленного вызова процедур, который позволяет приложениям в операционной системе удобно общаться друг с другом. D-Bus (The Desktop Bus) изначально создавался для взаимодействия пользовательских приложений в графической системе KDE.

D-Bus состоит из двух частей: низкоуровневого API и демона, выступающего в роли маршрутизатора сообщений. D-Bus предоставляет доступ к нескольким шинам: системной и сессионным. Системная шина предназначена для взаимодействия пользовательских приложений с операционной системой и любыми системными демонами. Сессионная шина — для пользовательских приложений, отдельная для каждого авторизованного пользователя.

В отличие от сокетов, которые поддерживаются любой POSIX-совместимой операционной системой, D-Bus не является стандартной компонентой, однако в прошивку контроллера он включен.

Сообщения в D-Bus бывают четырёх видов: вызовы методов, результаты вызовов методов, сигналы (широковещательные сообщения) и ошибки.

Концепция D-Bus включает в себя понятие объекта, сродни понятию объекта в ООП. Каждое приложение представляет из себя совокупность объектов. Получатели и отправители сообщений идентифицируются по адресу объекта, а каждый объект поддерживает несколько интерфейсов — группу именованных методов и сигналов. Также D-Bus предусматривает концепцию сервисов — уникальных местоположений приложения на шине. Сервисы позволяют запускать и останавливать привязанные к ним приложения по запросу.

Существуют высокоуровневые библиотеки для D-Bus практически для всех языков программирования и фреймворков, в частности, есть поддержка из Qt. Qt D-Bus [5] предоставляет удобную обертку над низкоуровневой моделью отправки сообщений; работу с сигналами, основанную на механизме сигналов и слотов в Qt; класс-адаптер для вынесения интерфейса обычных Qt-объектов на D-Bus шину.

Для вызова метода у удаленного объекта на шине можно воспользоваться одним из двух способов. Во-первых, можно последовательно создать вызов метода, отправить его, а потом получить и обработать результат. Во-вторых, (этот метод предоставляют высокоуровневые биндинги) для каждого удаленного объекта можно создать прокси-объект, обращение к которому будет выглядеть точно так же, как вызов метода у объекта ООП, но каждый «обычный» вызов метода будет конвертироваться биндингом в D-Bus вызов метода и отправляться на шину. Для того, чтобы можно было создать прокси-объект, сервис должен предоставить XML-файл определенного вида. В QT существует специальный класс — QtDBus Adaptor, с помощью которого для любого Qt объекта можно предоставить интерфейс «наружу», чтобы объект был доступен по D-Bus. По классу, созданному адаптером, и генирируется требуемый XML-файл (с использованием утилиты qdbuscpp2xml).

Из недостатков D-Bus стоит отметить, что отправка D-Bus сообщений требует больше вычислительных ресурсов по сравнению со стандартными средствами IPC, встроенными в ядро. Это происходит из-за большого числа переключений контекста между ядром и процессом-демоном, работающим на прикладном уровне. Были предприняты попытки перенести D-Bus в ядро (kdbus, Bus1), что должно было сильно повысить скорость его работы, но пока ни одна из реа-

лизаций не получила широкого распространения.

3.7. RabbitMQ

RabbitMQ [6] — приложение для распределенной передачи сообщений (его также называют message-oriented middleware – промежуточное программное обеспечение, ориентированное на обработку сообщений), которое позволяет взаимодействовать различным программам на основе стандарта AMQP. Представляет из себя брокер сообщений (message broker) — некий узел, который принимает сообщения от отправителей (producers) и направляет их получателям (consumers). Внутри RabbitMQ есть специальный буфер — очередь (queue), в которой хранятся сообщения. Очередь не имеет ограничений на количество сообщений, она в состоянии принять сколь угодно большое их количество — в сущности, это бесконечный буфер. Любое количество процессов может отправлять сообщения в одну очередь и получать сообщения из одной очереди.

Еще один важный элемент сервера RabbitMQ — коммутатор (exchange). Перед тем, как попасть в очередь, сообщения попадают в коммутатор, которые распределяет сообщения по очередям. Именно тип коммутатора определяет стратегию обмена сообщениями, например, будет ли сообщение направлено в очередь с определенным ключом (direct) или же всем видимым коммутатору очередям (fanout).

RabbitMQ простой в использовании, с гибкой системой маршрутизации и поддержкой всевозможных стандартов. Позволяет соединять получателей и отправителей, написанных на разных языках программирования, почти для каждого из которых уже реализован RabbitMQ клиент. Вместе с тем для текущей задачи это решение может оказать-

ся слишком медленным и громоздким, избыточным — «пушкой по воробьям». Кроме того, добавление RabbitMQ в прошивку контроллера приведет к увеличению размера прошивки, что нежелательно. Брокер сообщений в Rabbit написан на языке программирования Erlang, а это может усложнить процесс интеграции.

3.8. ZeroMQ

ZeroMQ [7] является легковесной и быстрой библиотекой для передачи сообщений. «Zero» в названии библиотеки подразумевает «zero broker» или же «zero latency» (нулевая задержка). В основе ZeroMQ лежат сокеты и не используется брокер, что делает это решение более быстрым, чем большинство приложений для передачи сообщений, использующих брокеры.

Некоторые особенности библиотеки описаны ниже.

1. Работа с вводом/выводом происходит асинхронно в фоновом режиме, не происходит никаких блокировок.

2. Можно настроить сеть так, что клиенты и серверы будут подключаться и отключаться в любое время и в произвольном порядке.

3. Сообщения сначала попадают во внутреннюю очередь, что позволяет не дожидаться конца отправки и даже соединения с сервером. ZeroMQ сам справляется с переполнением очередей, либо блокируя на время клиентов, либо отбрасывая сообщения, при этом максимальный размер очереди настраивается.

   Существенным отличием от RabbitMQ, кроме скорости передачи сообщений, является необходимость самостоятельно реализо-

вывать большую часть продвинутых возможностей (вроде гибкой маршрутизации и гарантированной доставки сообщений). Там, где в RabbitMQ можно обойтись всего парой строчек кода, в ZeroMQ приходится комбинировать различные части фреймворка.

3.9. nanomsg

Nanomsg [1] — это очень легковесная библиотека, написанная на языке программирования C, появившаяся из ZeroMQ и призванная исправить ее недостатки. Легковесность делает эту библиотеку подходящим решением для embedded-систем, для нее существуют биндинги для большинства популярных языков программирования, и она позволяет реализовать множество дизайн-паттернов взаимодействия. Однако, библиотека достаточно молодая и все еще не слишком хорошо документирована.

4. Тестирование

D-Bus уже добавлен в прошивку контроллера. Для оценки того, насколько он подходит для решения текущей задачи, была написана простая тестовая программа наподобие dbus-ping-pong [2].

Каждому сенсору сопоставим d-bus сервер, испускающий с определенной периодичностью широковещательные сигналы, а для программ, которым нужна информация от сенсора — d-bus клиент. Специально для тестирования в прошивку также была добавлена утилита dbus-monitor, стандартная для отлаживания приложений, использующих D-Bus. Для получения логов она была запущена на контроллере со следующими параметрами:

```
dbus-monitor --pcap >> filename.log
```

Полученные логи визуализировались и анализировались на компьютере с помощью программы Bustle, являющийся продвинутой версией dbus-monitor с графическим интерфейсом.

При увеличении нагрузки в определенный момент количество сообщений, которое фактически было отправлено по шине, перестало равняться количеству, которое должно было быть сгенерировано серверами – то есть процессор перестал справляться с нагрузкой. Стоит также учесть, что сама утилита мониторинга нагружает процессор пропорционально трафику на D-Bus шине, поэтому было решено использовать другую метрику.

В качестве метрики для оценки качества результата будем использовать показатель idle процессора — процент времени, в течение которого процессор выполняет специальный процесс «бездействия системы», который выполняется в случае, если не выполняется никакой

другой процесс. Для чистоты эксперимента отключим графический интерфейс на контроллере. Информацию будем получать с помощью системной утилиты dstat, запущенной со следующими параметрами:

```
dstat --cpu --output filename.csv
```

Смоделируем следующую нагрузку:

Сенсор	**Процессов**	**Сообщений в секунду**
Видеодатчик	1	30
Звуковой сенсор	1	30
Геймпад	1	10
Twitter	1	5
Ещё что-нибудь	1	5

Средний idle 35 %

Увеличим количество читателей у «высокочастотного» сенсора:

Сенсор	**Процессов**	**Сообщений в секунду**
Видеодатчик	2	30
Звуковой сенсор	1	30
Геймпад	1	10
Twitter	1	5
Ещё что-нибудь	1	5

Средний idle 24 %

Попробуем найти верхнюю границу по числу сообщений, при котором idle приемлем (около 20%). После нескольких экспериментов получим, при следующей конфигурации:

Сенсор	**Процессов**	**Сообщений в секунду**
Сенсор1	1	16
Сенсор2	1	16
Сенсор3	1	16
Сенсор4	1	16
Сенсор5	1	16

Средний idle 18 %

Стоит иметь в виду: в этих цифрах не учтено, что датчики сами по себе потребляют процессорное время. Однако известно, что работающие датчики суммарно снижают idle процессора не более чем на 15%. Таким образом, из результатов тестирования можно сделать вывод: D-Bus является удовлетворительным решением.

5. Архитектура

В ходе решения второй подзадачи... Было предложено 2 варианта решения задачи:

1. Сконструировать одну утилиту, которая будет заниматься маршрутизацией для всех сенсоров и клиентов.

2. Сконструировать адаптер для процесса-сенсора и запускать его отдельно для каждого сенсора.

С точки зрения скорости работы, первое решение выигрывает — множество запущенных процессов работают дольше из-за переключения контекста, возрастает нагрузка на память. С другой стороны, есть определенные преимущества и у второго решения — каждый процесс изолирован, и они не могут мешать друг другу. Кроме того, второе решение явно проще реализовать. Учитывая, что в конечном счете утилита, являющаяся оберткой над стандартным вводом и выводом, потребуется лишь приложениям, не интегрированным с D-Bus, а их будет не так много, возросшая нагрузка не критична.

Опишем второй вариант решения чуть подробнее. Для каждого процесса-сенсора надо реализовать адаптер, являющийся D-Bus сервером. Предполагается, что все процессы умеют общаться с «внешней средой» через стандартный ввод и вывод. В таком случае связь между изначальным процессом и соответствующим ему dbus-процессом можно будет организовать с помощью каналов стандартным для подобного рода утилит образом. Так, известно, что имеющиеся в распоряжении родительского процесса файловые дескрипторы копируются всем его дочерним процессам, что позволяет перенаправить стандартный ввод и вывод дочернего процесса в канал, заблаговременно

созданный из родительского процесса и доступный из него. Таким образом, когда кто-то захочет получить данные от сенсора, он сможет подписаться на сигнал от dbus-сервера, расположенного по известному адресу на шине. А чтобы отправить сенсору какие-то настройки или указания, достаточно сформировать dbus-сообщение или метод и разместить его на шине для нужного сервера.

Адаптер будет принимать на вход адрес на шине, по которому следует себя разместить, и процесс, который нужно запустить.

6. Заключение

В рамках курсовой работы были рассмотрены преимущества и недостатки использовавшегося ранее на контроллере решения на основе FIFO и составлен обзор других существующих решений задачи. В ходе работы были предприняты попытки разработать архитектуру утилиты, которая позволила бы избавиться от ряда недостатков FIFO, но от этой идеи пришлось отказаться — существуют более подходящие механизмы, на основе которых стоит проектировать решение, а также ряд уже существующих библиотек и приложений.

В конечном счете, было принято решение использовать D-Bus как средство межпроцессного взаимодействия. D-Bus имеет широкое распространение, хорошую документацию и доступ из фреймворка QT. Кроме того, D-Bus уже имеется в прошивке контроллера, что позволило быстро начать работу с ним. D-Bus был протестирован на контроллере, и в результате экспериментов сделан вывод, что решение на его основе вполне удовлетворяет существующим на данный момент требованиям.

В дальнейшем планируется более детально рассмотреть библиотеку nanomsg, поскольку, возможно, для организации IPC в контексте embedded систем она подходит больше.

Список литературы

[1] About nanomsg. — URL: http://nanomsg.org/.

[2] D-Bus Ping Pong Example. — URL: http://doc.qt.io/qt-5/qtdbus-pingpong-example.html.

[3] D-Bus freedesktop. — URL: https://www.freedesktop.org/wiki/Software/dbus/.

[4] Linux Programmer's Manual. unix(7). — URL: http://man7.org/linux/man-pages/man7/unix.7.html.

[5] Qt D-Bus.

[6] RabbitMQ official website. — URL: https://www.rabbitmq.com/.

[7] ZeroMQ official website. — URL: http://zeromq.org/.

[8] Робототехнический контроллер ТРИК. — URL: http://www.trikset.com/.