Распределённые системы. Семинары.

Бабичев С. Л., Коньков К. А.

Москва, 2008-2022

Удалённый вызов процедуры

Механизмы взаимодействия процессов:

- Сообщение, возможно, меняет контексты получателя
- Сообщение есть запрос на исполнение каких-то действий, ответ на который должен быть получен после совершения этих действий. Синхронизация происходит по факту получения ответного сообщения.

Удалённый вызов процедуры (`Remote Procedure Call, RPC`)
Вызывающая сторона запросом инициирует исполнение каких-либо действий на вызываемой. Вызывающая сторона исполняет роль *клиента*, а вызываемая — роль *сервера*.
С точки зрения клиента выдача запроса на исполнение функции на сервере выглядит как обыкновенный вызов обыкновенной функции.

Подсистема обработчика запросов

Клиент для выдачи запроса серверу (вызову серверной процедуры) должен знать:

- Идентификатор процедуры (проблема разрешения имён)
- Количество и типы аргументов процедуры
- Количество и типы возвращаемых значений.

Типичный код клиента.

```
bytevector bytes;
int64 offset = 0;
int32 size = 32768;
int32 processedBytes;
Error::Code code = Client.FileSystemObject.read(
  L"Media.flac", offset, size, bytes, processedBytes);
```

Пример: Microsoft RPC

Программа на IDL $\rightarrow$ Группа файлов реализации для клиента + Группа файлов реализации для сервера
Пример файла test.idl на языке MIDL:

```
[
  uuid (f7d60ee9-a80b-4b42-a2db-137ceb134eca),
  version(1.0),
  pointer_default(unique)
]
interface RemoteSignal
{
  error_status_t RegisterNotification(
    handle_t hBinding,
    [in, string] char * lpSignalName,
    [in, out] unsigned long* pnResult
  );
}
```

Пример: `Microsoft RPC` (продолжение)

`test.idl` $\rightarrow$ {`test_h.h` (77 строк) `test_c.c` (208 строк) `test_s.c` (210 строк)}
Минусы:

- один производитель ОС на клиенте и сервере
- чисто статический механизм. Невозможно расширить набор процедур без перекомпиляции
- сложно вызвать процедуру косвенно или по имени
- требуется сложная и часто избыточная инфраструктура

Передача данных при RPC: клиент

Вызов `FileSystemObject.read(L"Media.flac", offset, size, bytes, processedBytes);`
Метод `call` клиента:

- Сериализация идентификатора вызываемой функции
- Сериализация выходных аргументов
- Вызов транспортного уровня — передача серии
- Вызов транспортного уровня — приём серии
- Разбор серии — получение результатов
- Передача результатов исходному вызову

Общий цикл контекста сервера RPC

1. Порождение потоков-слушателей сетевых интерфейсов;
2. Привязка слушателя к объекту сервера (`listen`)
3. Слушатель ожидает соединения (`accept`)
4. Создание контекста клиента, создание клиентского потока в этом контексте, передаётся `Transport`.
5. Цикл работы клиентского потока

Передача данных при RPC: сервер

- Вызов транспортного уровня — приём входной серии
- Разбор серии — получение идентификатора процедуры
- Разбор серии — получение аргументов вызова
- Поиск вызываемой процедуры
- Вызов процедуры с нужными аргументами
- Получение результатов вызова
- Сериализация кодов ошибок и результатов вызова
- Вызов транспортного уровня — передача выходной серии

Выбор представления идентификатора процедуры

- Числовой (прямая адресация).
  - \+ Просто реализуется
  - \+ Эффективно
  - \- Малая гибкость и расширяемость
- `UID` (таблица преобразования)
  - \+ Высокая гибкость и расширяемость
  - \- Требуются внешние таблицы преобразования
- Символьный (таблица преобразования)
  - \+ Высокая гибкость и расширяемость
  - \- Требуется функция преобразования

Символьное представление идентификатора процедуры

Требуется преобразование
Имя процедуры $\rightarrow$ процедура
Варианты:

- Линейный поиск
- Бинарный поиск
- Поиск по дереву
- Поиск по хеш-таблице
- Ассоциативный поиск

Реализация хеш-таблицы

```
typedef int32 (*)(executorPtr)(void *context,
   Deserializer const &des, Serializer &ser);

struct hashItem {
  string name;
  executorPtr executor;
  ...
}

class ExecutorHashTable {
public:
  ExecutorHashTable(int32 initialSize);
  ~ExecutorHashTable();
  void feed(string const &name, executorPtr);
  executorPtr find(string const &name);
private:
  uint32 hash(string const &name);
  void resize(uint32 items);
  dispose(...);
};
```

Сервер: структура исполнителя

Пример: умножение двух 64-битных чисел без знака

```
int32 umul64(void *context, Deserializer const &des,
  Serializer &ser)
{
  uint64 a1 = des.getUint64();
  uint64 a2 = des.getUint64();
  ser.put(a1 * a2);
  return Error::OK;
}
```

Сервер: структура исполнителя

Пример: максимум любого количества 32-битных целых чисел

Передача аргументов только через сериализатор.

```
int32 maxInt32(void *context, Deserializer const &des,
  Serializer &ser)
{
  int32 max = 0;
  if (des.peekType() == SerialTypes::Int32Type)
  {
    max = des.getInt32();
    while (des.peekType == SerialTypes::Int32Type)
    {
      int32 x = des.getInt32();
      max = x > max ? x : max;
    }
  }
  ser.put(max);
  return Error::OK;
}
```

Подключение исполнителей к серверу

Заполнение хеш-таблицы ассоциаций:

```
mainHashTable.feed("umul64", umul64);
mainHashTable.feed("maxInt32",maxInt32);
```

Клиенты и клиентские вызовы серверных процедур

Используя метод call клиента вызываем удалённую процедуру umul64:

```
uint32 Client::umul64(uint64 arg1, uint64 arg2,
uint64 &result) {
    Serializer ser;
    ser.put(arg1);
    ser.put(arg2);
    Deserializer des;
    uint32 ret = call("umul64", ser, des);
    if (ret != Error::OK) {
        return ret;
    }
    res = des.getUint64();
    return Error::OK;
}
```

Главный цикл обработки клиентских запросов

```
bytevector inp;
string name;
executorPtr exec = mainHashTable.find(name);
if (exec != NULL) {
  Serializer ser(inp);
  Deserializer des;
  int32 code;
  try {
    code = exec(ser, des);
  }
  catch (Exception const &ex) {
    code = ex.getCode(); ...
  }
  if (code == Error::OK) {
    bytevector out = des.get();
  } else {
    // ^^d0^^9f^^d0^^b5^^d1^^80^^d0^^b5^^d0^^b4^^d0^^b0^^d0^^b5^^d0
  }
}
```

Передача массивов

```
RPC_read(int fd, void *buf, size_t size);
```

`buf` — указатель!

Что мы должны передать серверу?

Передача массивов

```
RPC_read(int fd, void *buf, size_t size);
```

`buf` — указатель!

Что мы должны передать серверу?

- Сам указатель как число?

Передача массивов

```
RPC_read(int fd, void *buf, size_t size);
```

`buf` — указатель!

Что мы должны передать серверу?

- Сам указатель как число?
- Факт, что аргумент — указатель?

Передача массивов

```
RPC_read(int fd, void *buf, size_t size);
```

`buf` — указатель!

Что мы должны передать серверу?

- Сам указатель как число?
- Факт, что аргумент — указатель?
- Указатели мы передать не можем!

Что делать?

Использовать механизм реплик.

Алгоритм передача массивов

1. Клиент передаёт запрос на операцию чтения массива в `stub`, адрес `buf` корректен!
2. `stub` сохраняет указатель на блок памяти и размер
3. `stub` формирует серию с именем `"RPC_read"` и `size`
4. `stub` вызывает `send`
5. `Server`: `receive`
6. `Server`: десериализация запроса — имя и аргументы в виде серии.
7. `Server`: находится процедура, ей передаётся серия аргументов (в том числе `size`)
8. Процедура: обеспечивает `size` байт для чтения.
9. Процедура: заполнение блока.
10. Процедура: выходная серия, `readsize`, `readsize` байт.
11. `Server`: `send`
12. `stub`: копирует данные из серии в буфер `buf`.

Исполнение удалённого кода

Клиент желает исполнить код. Исполняемый код заранее неизвестен. Как расширить RPC-сервер?

- Через интерпретатор машинно-независимого кода
- Через компилируемый код

расширение RPC: интерпретация

Метод интерпретации

- Клиент преобразовывает код в байт-код
- Код передаётся на сервер предопределённым вызовом RPC
- Код регистрируется на сервере (`feed`)
- Клиент вызывает RPC, исполняющую код

расширение RPC: компиляция

Метод компиляции

- Клиент запрашивает архитектуру сервера и версию ОС
- Клиент компилирует код
- Клиент вызывает RPC для передачи исполнимой библиотеки
- Клиент вызывает RPC для регистрации библиотеки. Сервер обнаруживает точки входа и добавляет их в хеш-таблицу ассоциаций.
- Клиент вызывает RPC, исполняющую код

Достоинства и недостатки этих подходов.
Интерпретация машинно-независима, но медленная.
Компиляция производительна, но не всегда реализуема.

Ограничения при реализации удалённо исполняемого кода

Исполнимый код должен быть реализован в виде динамически загружаемых библиотек. Ограничения:

- компоновка с адресным пространством сервера
- невозможность использования глобальных объектов сервера
- самодостаточность кода, отсутствие неразрешённых внешних ссылок
- использование указателей и ссылок между адресными пространствами библиотеки и сервера