УДК 519.687

ИСПОЛЬЗОВАНИЕ МЕХАНИЗМОВ УДАЛЕННОГО ВЫЗОВА ПРОЦЕДУР ПРИ ОТЛАДКЕ И ТЕСТИРОВАНИИ ВСТРАИВАЕМЫХ СИСТЕМ КОСМИЧЕСКОГО НАЗНАЧЕНИЯ

А. В. Лапин, Т. А. Мадумаров

При разработке и отладке программного обеспечения для бортовой аппаратуры, состоящей из множества как программных, так и аппаратных компонентов, разнесённых по разным узлам сети, в определённый момент возникает необходимость провести отладку всего комплекса целиком. Для успешного выполнения подобной задачи требуется реализовать в коде программы протокол удалённого вызова процедур, который позволит обращаться к удалённым узлам системы и исполнять на них код таким образом, будто вызов процедуры произошёл непосредственно из самого узла. В работе представлено сравнение существующих протоколов удалённого вызова процедур, проведён их сравнительный анализ и нагрузочные тесты. Показано, в каких случаях требуется разработка собственной имплементации протокола, а в каких можно положиться на уже готовое штатное решение. Приведены качественные и количественные результаты сравнения описываемых протоколов.

***Ключевые слова**: встраиваемые системы, распределённые системы, удалённый вызов процедур.*

Встраиваемые системы космического назначения часто представляют собой вычислительные сети, каждый узел которой называется вычислительным модулем (ВМ). Комплексной отладкой встраиваемой системы будем называть такую отладку, при которой отлаживается более одного ВМ. Средой комплексной отладки будем называть такой комплекс программного и аппаратного обеспечения, который необходим для автоматизированного тестирования и отладки встраиваемых систем. Среда комплексной отладки, разработанная в АО «НИИ «Субмикрон» [1], используется для отладки и тестирования как аппаратных, так и виртуальных прототипов встраиваемых систем и состоит из нескольких клиентов отладчика *GNU Debugger* (*GDB*), одного многоканального *GDB*-сервера, модулей управления стендовой аппаратурой, а также низкоуровневого отладочного сервера, который осуществляет непосредственное отладочное взаимодействие с целевым устройством (ЦУ) или его моделью (Рис. 1).

Алгоритм большинства автоматизированных тестов или скриптов отладки построен следующим образом:

1. Настройка стендовой аппаратуры, например, измерительных приборов, источников питания и т. п.
2. Загрузка отлаживаемой программы в ЦУ.
3. Предварительная настройка отладчика, установка точек останова и т. п.
4. Запуск отлаживаемой программы.
5. Ожидание завершения программы или выхода на точку останова.
6. Чтение и анализ показаний измерительной аппаратуры.

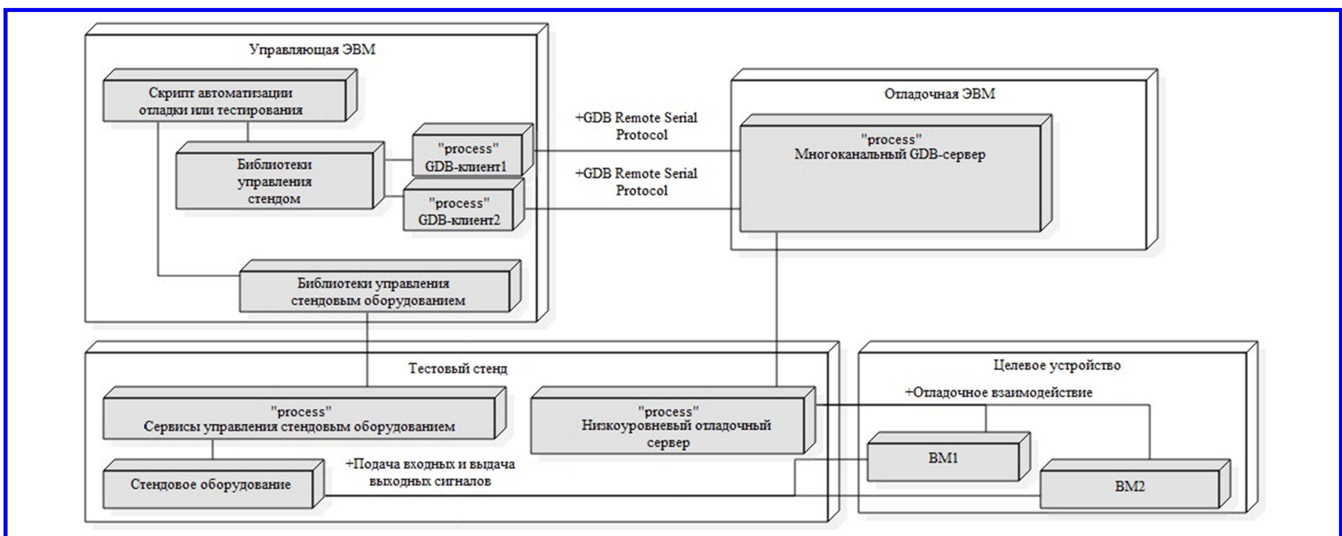

Рис. 1. Диаграмма развертывания среды комплексной отладки

Для осуществления пунктов 1 и 6 необходима организация автоматизированного управления стендовой аппаратурой, а для пунктов 3, 4, 5 – автоматизированного управления отладчиком. Загрузка программы в целевой прибор (пункт 2) может осуществляться разными способами: как при помощи отладчика, так и при помощи специальных программаторов, входящих в состав стенда.

Автоматизация данного алгоритма осуществляется с использованием скриптовых языков, таких как Python или JavaScript. Зачастую у заказчика уже есть какая-то система моделирования или управления тестированием и отладкой всего космического аппарата, в которую система комплексной отладки отдельного прибора должна встраиваться с минимальными трудозатратами.

Таким образом, появляется два контура управления средой комплексной отладки: управление отладчиками и управление стендовым оборудованием. В обоих из этих контуров управления применяются механизмы удаленного вызова процедур (remote procedure call, *RPC*). При этом данные контуры принципиально различаются по следующим признакам:

изменчивость: в процессе разработки и эксплуатации прибора состав тестового стенда может поменяться. Для одного и того же прибора может существовать несколько вариантов тестового стенда. Протокол взаимодействия *GDB*-сервера с моделью целевой аппаратуры или отлаживаемым прибором меняется относительно редко, поскольку сам отладчик очень стабилен;

кроссплатформенность: управление стендовым оборудованием может осуществляться с применением различных скриптовых языков программирования (ЯП), поскольку архитектурно этот уровень расположен ближе к интерпретатору скриптов, а последний может быть выбран заказчиком. Взаимодействие между многоканальным *GDB*-сервером и сервером низкоуровневой отладки целиком скрыто от пользователя, поэтому они оба были разработаны на одном ЯП – C++;

быстродействие: как правило, для управления стендовой аппаратурой редко необходимо передавать большие объемы данных. Через отладочный канал может осуществляться прошивка прибора непосредственно в память ЦУ, что несколько увеличивает требования к быстродействию.

Существует большое количество протоколов *RPC*, например:

SOAP (*Simple Object Access Protocol* – простой протокол доступа к объектам) – был впервые опубликован в конце 90-х годов прошлого века [2] и использовал формат XML в качестве средства обмена информацией между клиентом и сервером;

REST (*Representational State Transfer* – передача репрезентативного состояния), впервые описанный в 2000 году [3]. Как и *SOAP*, *REST* опирается на использование стандартного транспортного протокола HTTP. *REST* построен на принципе *CRUD* (*Create*, *Read*, *Update*, *Delete*) и использует соответствующие HTTP-запросы для манипулирования данными;

JSON-RPC [4] – более гибкий и легковесный протокол, чем *REST*. Он является, во-первых, менее формализованным, а, во-вторых, не опирается на принцип *CRUD*, по сути, используя только метод *POST*. Также *JSON-RPC*, начиная с версии 2.0, поддерживает пакетный вызов нескольких методов за один HTTP-запрос;

gRPC [5] и *Qt Remote Objects* [6] – данные протоколы используют генерацию кода и бинарные форматы передачи данных непосредственно через протокол *TCP* (*Transmission Control Protocol* — протокол управления передачей) и, как следствие, они должны быть быстрее всех предыдущих.

Поскольку АО «НИИ «Субмикрон» уже широко использует *Qt* в качестве основного каркаса для разработки больших проектов на C++, было решено не использовать *gRPC*, так как в *Qt* уже есть интегрированный аналог, использующий похожий принцип.

Для осуществления *RPC* через *Qt Remote Object* необходимо произвести следующие действия:

1. Описать интерфейс в *repc*-файле. Это специальный формат файла, который предназначен для генерации кода. Например:

```
// файл foo.repc
class Foo
{
  SLOT(int bar(int, const QString&));
};
```

2. Разработать серверную часть. Итогом генерации кода будет абстрактный класс источника (*source*), чистые виртуальные методы которого необходимо будет реализовать. В данном контексте источник – это класс, который экспортирует необходимый функционал из сервера. Например:

```
// файл server/foo.hpp
#include “rep_foo_source.h”
class Foo: public FooSimpleSource
{
Q_OBJECT
public:
  Foo(QObject* parent = nullptr);
  int bar(int a, const QString& b);
};
```

3. Разрешить экспортирование методов класса из сервера путем вызова метода *QRemoteObjectHost::enableRemoting*, например:

```
// файл server/main.cpp
#include "foo.hpp"
int main(int argc, char** argv) {
  …
        Foo foo;
        QRemoteObjectHost node("local:foo");
        node.connectToNode("local:switch"));
        node.enableRemoting(&foo);
        …
}
```

4. В клиентской части необходимо воспользоваться другим сгенерированным файлом – репликой (*replica*). Объект-реплика отвечает за импортирование функционала в клиент. Источник и реплика имеют один и тот же интерфейс. Например:

```
// файл client/main.cpp
#include "rep_foo_replica.h"
int main(int argc, char** argv) {
  …
         QRemoteObjectNode node;
         node.connectToNode("local:foo"));
         QSharedPointer<FooReplica> foo_ptr;
         foo_ptr.reset(node.acquire<FooReplica>());
         …
     }
```

5. Вызывать нужный метод из объекта-реплики, например:

```
// Где-то далее
int res = foo_ptr.bar(3, "Hello, word!");
```

Из примеров видны очевидные плюсы данного протокола. Во-первых, интерфейс взаимодействия между клиентом и сервером формализован в виде *repc*-файлов. Во-вторых, весь обмен информацией по сети скрыт от пользователя в сгенерированных файлах. Из этого следует также, что поскольку у реплики и источника одинаковый интерфейс, удаленный объект в случае необходимости легко подменяется на локальный и наоборот.

Из минусов протокола *Qt Remote Object* следует отметить слабую гибкость: изменение интерфейса будет требовать повторной генерации кода и внесения изменения в код всех клиентов. Для того чтобы этого избежать, существуют динамические источники (*dynamic source*), но их использование не столь удобно и «бесшовно». Поскольку протокол использует бинарные данные, его отладка затруднена. Также важным минусом является отсутствие кроссплатформенности: для использования данного протокола и в клиентской, и в серверной части необходимо использование библиотеки *Qt*.

Протоколы *RPC*, использующие текстовые форматы для передачи информации об удаленных процедурах, как правило, являются кроссплатформенными и не требуют генерации кода. Популярными протоколами такого типа являются *SOAP*, *REST* и *JSON-RPC*.

Протокол *SOAP* – это проверенный временем стандарт, который на данный момент выглядит несколько архаичным, поскольку содержит излишне много вспомогательной информации, но все еще поддерживается разработчиками. Существуют библиотеки готовых реализаций этого протокола для многих популярных ЯП, включая C++, Python и JavaScript.

Протокол *SOAP* использует формат *XML* для сериализации данных. Несмотря на то, что данный формат является текстовым, он является менее удобным для человеческого восприятия, чем формат *JSON*. Протокол *REST* может использовать как *JSON*, так и *XML*, он создавался скорее с целью манипуляции данными на удаленном сервере и для того, чтобы разработать *RESTful API*, необходимо поддерживать принцип *CRUD*.

Протокол *JSON-RPC* использует *JSON* в качестве формата передачи данных. Например:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42,
23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}
```

Данный пример иллюстрирует вызов метода и ответ сервера. Каждая транзакция представляет собой *JSON*-объект со следующими полями:

- *jsonrpc* (строка) – версия протокола *JSON-RPC*. Номер версии должен поддерживаться сервером, иначе ответом от него будет сообщение об ошибке;
- *method* (строка) – текстовое название метода. Если сервер не реализует такой метод, то он вернет сообщение об ошибке;
- *params* (список или объект) – аргументы удаленной процедуры;
- *id* (целочисленное значение) – идентификатор запроса. Ответная транзакция для данного запроса будет иметь такой же идентификатор.

Ответная транзакция может отсутствовать, если удаленная процедура не возвращает значения, иметь поле *result*, содержащее возвращаемое значение, либо поле *error*, содержащее код и сообщение об ошибке.

Протокол *JSON-RPC* поддерживает пакетный вызов нескольких функций за один запрос, например:

```
    --> [
        {"jsonrpc": "2.0", "method": "sum", "params":
[1,2,4], "id": "1"},
        {"jsonrpc": "2.0", "method": "subtract", "params":
[42,23], "id": "2"},
    ]
<-- [
        {"jsonrpc": "2.0", "result": 7, "id": "1"},
        {"jsonrpc": "2.0", "result": 19, "id": "2"},
    ]
```

Библиотеки, упрощающие разработку как серверов, так и клиентов *JSON-RPC*, существуют для многих ЯП, включая Python, JavaScript и C++. Для удобства разработки *JSON-RPC*-серверов на *C++* с использованием библиотеки *Qt* была разработана библиотека *sm_qt_json_rpc*, которая использует встроенную в *Qt* систему *Qt Meta-Object System* [7] для оборачивания объектов и функций «на лету». С помощью макроса *Q_INVOKABLE* можно пометить метод, как доступный по строковому названию во время исполнения. Такой же принцип используется в библиотеке *Qt Remote Objects*. Такое решение обладает высокой кроссплатформенностью, большой гибкостью и не требует генерации кода на стороне клиента. В качестве *HTTP*-сервера используется библиотека *QtHttpServer*.

Например, пусть имеется класс *Foo*, который имеет *Q_INVOKABLE*-метод *bar*:

```
    // файл foo.hpp
class Foo: QObject {
Q_OBJECT
public:
  Foo(QObject* parent);
  Q_INVOKABLE int bar(int a, const QString& b);
};
```

Для того, чтобы *JSON-RPC*-сервер экспортировал метод данного класса, необходимо написать очень простой код:

```
// файл main.cpp
#include "foo.hpp"
#include <sm_qt_json_rpc_server/json_rpc_route.hpp>
#include <QtHttpServer>
#include <functional>

int main(int argc, char** argv) {
    …
    QCoreApplication app(argc, argv);

    QHttpServer server;
    Foo foo(nullptr);
    JSONRPCRoute route(&fake);

    server.route("/foo/",
                 [&route](const QHttpServerRequest& req,
                 QHttpServerResponder&& responder) {
        fake_route.processRequest(req, std::move(responder));
    });
    return app.exec();
}
```

Данный пример создает класс *sm_qt_json_rpc::server::JSONRPCRoute*, который при помощи *Qt Meta-Object System* находит все *Q_INVOKABLE*-методы класса *Foo* и оборачивает их в функции, которые принимают *JSON*-данные. Классы для разработки *JSON-RPC*-клиентов также присутствуют в данной библиотеке. Пусть сервер слушает порт 13370, тогда код подключения клиента к серверу будет выглядеть следующим образом:

```
JSONRPCConnection connection;
connection.connectToRPCServer(QUrl(
    "http://localhost:13370/fake_object/"));
```

После этого необходимо получить указатель на объект метода. Если предполагается использовать *RPC* в синхронном режиме, ожидая окончание каждого обмена, необходимо воспользоваться методом *JSONRPCConnection::wrappedMethod*. Для работы в асинхронном режиме необходимо воспользоваться методом *JSONRPCConnection::wrappedAsyncMethod*, который возвращает объект *QFuture*:

```
auto bar = connection.wrappedMethod<
  int, int, const QString&>("bar");
int res = bar(3, "Hello!");
```

Библиотека *sm_qt_json_rpc* упрощает использование протокола *JSON-RPC* при разработке программного обеспечения с использованием библиотеки *Qt*.

Для оценки производительности было произведено несколько замеров производительности, которые сравнивали *JSON RPC*, *Qt Remote Objects* и в качестве эталона – локальные вызовы функций. Для замеров производительности использовалась библиотека *QTestLib* [8], входящая в состав каркаса *Qt*. Производился замер производительности функций со строковыми (рис. 2) целочисленными параметрами (рис. 3) и параметром в виде массива «сырых» бинарных данных (типа *QByteArray*, рис. 4). Результаты измерения показали, что использование библиотеки *sm_qt_json_rpc* медленнее, чем *Qt Remote Objects*, особенно при передаче больших массивов данных, что связано с тем, что протокол *JSON-RPC* – текстовый, а *Qt Remote Objects* – бинарный.

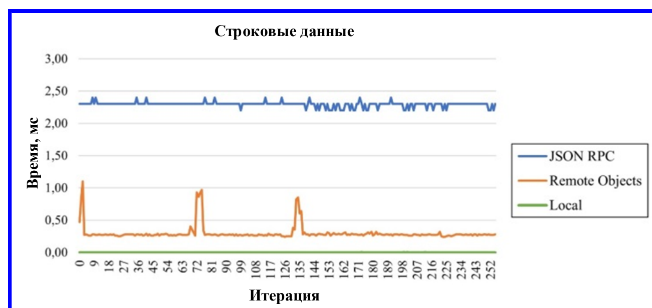

Рис. 2. Анализ производительности передачи строковых данных

Рис. 3. Анализ производительности передачи целочисленных данных

Рис. 4. Анализ производительности передачи байтового массива

Введем коэффициент замедления $D = \overline{T_{Jsonrpc}} / \overline{T_{qt}}$, где $\overline{T_{Jsonrpc}}$ и $\overline{T_{qt}}$ – это среднее время вызова одной удаленной процедуры с использованием библиотек *sm_qt_json_rpc* и *Qt Remote Objects* соответственно. Для строковых данных D_{str} = 2,43, а для целочисленных: D_{int} = 5,54 (табл. 1).

Для больших массивов данных данная величина колеблется от 8,5 для 1 Кбайт данных до 69 для 256 Кбайт. Это связано с тем, что при передаче бинарных данных в формате *JSON* используется стандарт *base64*, который дает накладные расходы по размеру данных примерно в 33%. Помимо всего прочего, сам перевод бинарных данных в строку и обратно также несколько замедляет вызов функций. Из результатов данного замера можно сделать вывод, что *JSON-RPC* совершенно не подходит для передачи больших объемов бинарных данных (табл. 2).

Таблица 1

Коэффициент замедления для целочисленных и строковых данных

Тип данных	$\overline{T_{Jsonrpc}}$, мс	$\overline{T_{qt}}$, мс	*D*
Строковый	2,29	0,94	2,43
Целочисленный	2,27	0,41	5,54

Таблица 2

Зависимость коэффициента замедления от размера бинарных данных

Размер, байт	$\overline{T_{Jsonrpc}}$, мс	$\overline{T_{qt}}$, мс	*D*
1024	2,4	0,28	8,57
10240	3,4	0,29	11,72
102400	14	0,35	40
262144	32	0,46	69,56

Для автоматизации управления тестовыми стендами был выбран протокол *JSON-RPC*. В том числе при разработке серверов стендовой аппаратуры на ЯП *C++* используется библиотека *sm_qt_json_rpc*. Несмотря на то, что данное решение является менее производительным, оно более гибкое и устойчивое к изменениям, а также более кросс-платформенное. В данном случае более низкая производительность не перевешивает архитектурных плюсов.

Для связи между многоканальным *GDB*-сервером и сервером низкоуровневой отладки был выбран протокол *Qt Remote Objects*. Это связано с тем, что более низкая гибкость данного протокола не играет большой роли, ведь набор отладочных функций не изменяется со временем. При этом данный протокол, наоборот, легче интегрируется в приложения с использованием *Qt*, его проще тестировать, а также высокая производительность при больших объемах данных позволит быстрее загружать программное обеспечение в целевое устройство.

Литература

1. Среда для комплексной отладки многомашинных встраиваемых систем космического назначения на основе GDB / Т. А. Мадумаров, А. В. Стефанцов, А. В. Лапин [и др.] // Наноиндустрия. – 2020. – 13(S5-1). – С. 139–140.
2. Simple Object Access Protocol (SOAP) 1.1. – Текст : электронный // W3C : [сайт]. – URL : https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
3. Fielding R. T. Architectural styles and the design of network-based software architectures : dissertation submitted in partial satisfaction of the requirements for the degree of doctor of philosophy in Information and Computer Science / R. T. Fielding ; University of California. – Irvine, 2000. – [180] p.
4. JSON-RPC 2.0 Specification // JSON-RPC Working Group. – 2012. – URL : https://www.jsonrpc.org/specification#extensions.
5. gRPC : website. – 2023. – URL : https://grpc.io/.
6. Qt Remote Objects // Qt Group : website. – 2023. – URL : https://doc.qt.io/qt-6/qtremoteobjects-index.html.
7. The Meta-Object System // Qt Group : website. – 2023. – URL : https://doc.qt.io/qt-6/metaobjects.html.
8. Qt Test // Qt Group : website. – 2023. – URL : https://doc.qt.io/qt-6/qttest-index.html.

Поступила в редакцию 27.09.2023

***Александр Владимирович Лапин**, ведущий инженер-программист, т. 8 (499) 731-89-31, e-mail: xanderius@mail.ru.*
***Талгат Асхатович Мадумаров**, начальник лаборатории, e-mail: madrook00@gmail.com.*
(АО «НИИ «Субмикрон»).

APPLICATION OF REMOTE PROCEDURES CALLING MECHANISMS DURING DEBUGGING AND TESTING OF AEROSPACE EMBEDDED SYSTEMS

A. V. Lapin, T. A. Madumarov

During development and debugging of software for onboard devices which can contain multiple hardware and software components there could be a task to debug the system as whole. Such components may be spaced apart as different network nodes. For successful implementation of described task there are some remote procedures calling protocol should be described in code. This approach allows to execute procedures on remote target like it was called directly from remote node itself. This paper represents a comparison of existing RPC protocols based on benchmark testing. It is showed which of them may be used as is in described task implementation and in which case we needed to develop our own variation of protocol. Benchmark results are provided.

Key words: *embedded systems, distributed systems, remote procedure calling.*

References

1. IDE for complex debugging of multimachine aerospace embedded systems based on GDB / T. A. Madumarov, A. V. Stefantsov, A. V. Lapin [et. al.] // Nanoindustria. – 2020. – 13(S5-1). – P. 139–140.
2. Simple Object Access Protocol (SOAP) 1.1. – Text : electronic // W3C : [website]. – URL : https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
3. Fielding R. T. Architectural styles and the design of network-based software architectures : dissertation submitted in partial satisfaction of the requirements for the degree of doctor of philosophy in Information and Computer Science / R. T. Fielding ; University of California. – Irvine, 2000. – [180] p.
4. JSON-RPC 2.0 Specification // JSON-RPC Working Group. – 2012. – URL : https://www.jsonrpc.org/ specification#extensions.
5. gRPC : website. – 2023. – URL : https://grpc.io/.
6. Qt Remote Objects // Qt Group : website. – 2023. – URL : https://doc.qt.io/qt-6/qtremoteobjects-index.html.
7. The Meta-Object System // Qt Group : website. – 2023. – URL : https://doc.qt.io/qt-6/metaobjects.html.
8. Qt Test // Qt Group : website. – 2023. – URL : https://doc.qt.io/qt-6/qttest-index.html.

Alexander Vladimirovitch Lapin*, senior software developer,*
t. 8 (499) 731-89-31, e-mail: xanderius@mail.ru.
Talgat Askhatovitch Madumarov*, team leader, e-mail: madrook00@gmail.com.*
(SC «RSC «Submicron»).