А.Н. Ответчиков[1]

МЕХАНИЗМЫ ВЗАИМОДЕЙСТВИЯ ПРОГРАММ И ПРОГРАММНЫХ КОМПЛЕКСОВ

В статье дается краткий обзор существующих механизмов взаимодействия программ и программных комплексов с выделением их ключевых особенностей, достоинств и недостатков.

***Ключевые слова**: повторное использование кода, интероперабельность, программный интерфейс, межпроцессное взаимодействие, сервисно ориентированная архитектура.*

A.N. Otvetchikov

PROGRAM INTERFACE AND INTERFACE OF PROGRAM SYSTEMS

This article gives a brief overview of existing mechanisms of interaction of programs and software systems with separation of their key features, advantages and disadvantages.

***Keywords**: code reuse, interoperability, software interface, inter-process communication, service-oriented architecture.*

Современные компьютеры давно уже перестали быть «вещью в себе», отдельными устройствами с законченной функциональностью. Разнообразие платформ и программ для них позволяет использовать компьютер во многих областях человеческой деятельности. Однако при создании сложных программных комплексов оказывается, что значительная часть работы заключается в решении уже реализованных ранее задач. Чтобы избежать непродуктивных трудозатрат на «изобретение велосипеда» – многократного повторения уже написанного, – при разработке широко используется методология *повторного использования кода* (code reuse), заключающаяся в том, что система строится из уже готовых, написанных ранее частей – «кирпичиков», при этом «кирпичики» должны точно подходить друг к другу как части одной головоломки, то есть между ними должна обеспечиваться *интероперабельность*.

Первый и наиболее широко распространенный способ повторного использования кода заключается в механизме библиотек. В этом случае модули, которые можно использовать повторно, выделяются в специальные контейнеры – библиотеки, которые подключаются к основной программе либо на этапе разработки (статические библиотеки), либо на этапе выполнения (динамические библиотеки). Интероперабельность между отдельными библиотеками и основной программой обеспечивается единством используемого программного интерфейса (*Application Programming Interface, API*).

Следует заметить, что кроме технической возможности повторного использования кода – *технической интероперабельности* – необходимо обеспечивать также «взаимопонимаемость» информации, передаваемой модулю или принимаемой от него – *семантическую интероперабельность*. В случае с библиотекой забота о семантической интероперабельности лежит на ее авторе и достигается путем предоставления сведений (документации) о типах, возможных значениях и смысле параметров вызываемых функций и их результата.

Второй способ заключается в предоставлении собственно исходного кода той или иной функции для включения в текст программы. В отличие от библиотек, где хранятся уже транслированные, платформо зависимые модули, исходный код обычно *портабелен*, то есть при сравнительно небольших изменениях он способен работать на различных платформах. Особенное распространение этот способ получил при объектно ориентированном подходе: разработчику предлагается набор неких объектов, наследуя и модифицируя которые он может построить сложную систему с минимумом собственного кода.

Здесь уместно упомянуть еще об одном аспекте – *правовой интероперабельности*. Каждый блок «чужого» кода, внедренного в программу, имеет своего автора и условия распространения, то есть лицензию. Если, как правило, отдельные ограничения на использование библиотек не касаются основной программы (разумеется, при этом должны соблюдаться положения лицензии на саму библиотеку или продукт, ее включающий), то в случае прямого использования кодовой базы

[1] Аспирант НОУ ВПО «Российский новый университет».

могут возникнуть проблемы с лицензированием. Например, согласно условиям популярной лицензии *GPL*, нельзя использовать попадающий под ее действие код в составе продуктов, распространяющихся без исходных текстов.

Описанные способы хороши, так как требуют минимум трудозатрат программиста, быстры в исполнении, но имеют фундаментальные недостатки:

• во-первых, семантическая интероперабельность обеспечивается авторами библиотек или блоков кода так, как они сами это понимают: отсутствует стандартизация – унификация семантики;

• во-вторых, использование может быть неудобно для программиста: набор функций конкретных библиотек может быть недостаточен для данной программы, или, наоборот, слишком широк, или же логика реализации той или иной задачи может не совпадать с логикой основной программы;

• в-третьих, слабая *адаптабельность* – семантическая интероперабельность всех компонентов согласуется на этапе разработки программы: программа «привязана» к конкретной версии библиотеки или блока кода, в случае их модификации требуется доработка;

• в-четвертых, *безынициативность*: библиотечные функции и блоки кода выполняются только по инициативе основной программы и только совместно с ней (в ее *контексте*). Это делает невозможным построение из них сложных систем со своей внутренней логикой, реакцией на события и с сохранением своего контекста;

• в-пятых, все модули вместе с основной программой должны быть сосредоточены на одной физической системе (компьютере, кластере) – в одном *пространстве адресов*.

Для исключения или, по крайней мере, смягчения последнего недостатка, с появлением и широким развитием сетевых технологий появились механизмы *удаленного вызова процедур* (*Remote Procedure Call, RPC*). Сохраняя принципы работы с библиотеками – передача управления и данных с приостановкой работы до ожидания ответа – механизм *RPC* позволяет вызывать процедуры и функции в другом адресном пространстве, то есть на удаленных системах.

Существует множество реализаций механизма *RPC* для различных платформ и задач: *Sun RPC* (*RFC*-1831), *Microsoft RPC*, *NET Remoting*. Они имеют различную архитектуру и существенно различаются по своим возможностям. На основе механизма *RPC* построены многие сетевые файловые системы: *NFS* (*Sun RPC*), *SMB* (*MS RPC*) и другие. Однако такие проблемы, как безынициативность, слабая адаптабельность и неунифицированная семантика данный механизм не устраняет.

Чтобы побороть проблему безынициативности, следует окончательно отказаться от парадигмы «начальник» (инициатор) – «подчиненный» (исполнитель), уже частично демонтированной в объектно ориентированном подходе, где каждый объект может быть и инициатором, и исполнителем, и перейти к парадигме вычислительно не зависимых друг от друга модулей, объединенных посредством межпроцессного (межпотокового, межпрограммного) взаимодействия – *Inter-Process* (*Inter-Thread*, *Inter-Application*) *Communication*, *IPC*.

Существует множество вариантов межпроцессного взаимодействия, ориентированных на передачу управляющих сообщений (*MS Message Loop*, *UNIX SysV*), передачу данных (сокеты, анонимные и именованные каналы, разделяемая память), синхронизации (семафоры).

Взаимодействие компонентов, расположенных в рамках одной системы (компьютера, кластера), отличается тем, что влияние линии связи не учитывается. Подразумевается, что связь идеальна. Некоторые механизмы *IPC* учитывают работу транспортного уровня, в частности позволяют указывать адреса целевых систем и обрабатывать нештатные ситуации, возникающие при работе сети. Такие механизмы называют *толерантными* к сети.

Реализация простейших механизмов, в частности анонимных и именованных каналов (*pipelines*) и сокетов (*sockets*), платформо зависима, например в ОС семейства *Windows* каналы создаются в специальной файловой системе, в то время как в *UNIX*-подобных ОС их можно располагать в произвольном месте ФС или в памяти. Эти механизмы не предусматривают семантической интероперабельности (отправителю и получателю данных приходится согласовывать их содержание и значение самостоятельно, обычно на уровне разработки программ по документации) и *сериализацию* (то есть унификацию представления для разных систем) данных.

Более развитые способы *IPC* предусматривают сериализацию и предъявляют определенные требования к семантике. В этом случае говорят о межпрограммном *интерфейсе*. Примером могут служить протоколы работы сетевых СУБД (*MySQL*, *PostgreSQL*, *MS SQL*). К сожалению, развитость этих интерфейсов не преодолевает проблемы слабой адаптабельности и безынициативности. Кроме того, как правило, развитые интерфейсы не обладают универсальностью, а, напро-

тив, узкоспециализированны, как те же интерфейсы СУБД.

В настоящее время организация по распространению открытых стандартов структурированной информации (*OASIS*) активно продвигает концепцию сервисно ориентированной архитектуры (*SOA*, *service-oriented architecture*), согласно которой программная система состоит из отдельных сервисов (служб), распределенных по узлам сети. Совместное функционирование всех сервисов обеспечивается стандартизацией их интерфейсов: в части технической интероперабельности это достигается использованием распространенных и хорошо документированных протоколов, механизмов, средств и способов представления: *DCOM*, *CORBA*, *SOAP*, *WSDL*, *XML* и др., в части семантической интероперабельности требования и рекомендации слабее.

Наиболее значителен успех *SOA* в построении так называемых *web*-сервисов: неких программных систем, идентифицируемых через *Uniform Resource Identifier* (*URI*) и имеющих *WSDL*-интерфейсы для работы с *XML*-сериализованными данными.

При всех очевидных достоинствах (универсальность, инициативность всех компонентов, адаптабельность, простота реализации) главный недостаток таких систем – низкая скорость работы и ресурсоемкость, связанны, в первую очередь, с применением достаточно громоздкого способа сериализации – *XML*, а также ограничения, связанные с использованием изначально не рассчитанных на данный класс задач интернет-протоколов, в частности *http*.

Также концепцию *SOA* реализуют сравнительно новые межпрограммные интерфейсы – *D-BUS* и *Plan9*, свободные от данных недостатков. Интерфейс *D-BUS* нашел широкое применение в *UNIX* подобных ОС. С его помощью осуществляется взаимодействие таких сложных программных комплексов, как оболочки *GNOME* и *KDE*, механизм автоматического подключения устройств *HAL*. Интерфейс *Plan9*, реализующий концепцию «все через файл», имеет большой потенциал, но используется сравнительно редко в силу своей специфичности.

Подводя итоги, можно предположить, что основная работа по совершенствованию межпроцессного взаимодействия, скорее всего, будет направлена на бóльшую универсализацию и повышение адаптабельности взаимодействующих систем за счет стандартизации семантики. Возможен отказ от громоздких способов сериализации данных в пользу более легковесных и менее требовательных к вычислительным ресурсам. Наверняка «рабочая лошадка» – протокол *http* – будет заменен более универсальным и специализированным. Интересным вектором развития концепции *SOA* видится отказ от иерархического способа доступа к объектам в пользу семантического или смешанного. Но это – дело ближайшего будущего.

Литература

1. Немет, Э., Снайдер, Г., Хейн, Т. Руководство администратора Linux. – Вильямс, 2005. – 880 с.

2. Харт, Дж. М. Системное программирование в среде Windows. – Вильямс, 2005. – 592 с.

3. Википедия – свободная энциклопедия [электронный ресурс] – http://ru.wikipedia.org, режим доступа свободный.

4. Открытые системы: Фейгин, Д. Концепция SOA [электронный ресурс] – http://www.osp.ru/os/2004/06/184447/_p1.html , режим доступа свободный.