УДК 004.738 DOI: 10.17587/it.24.393-401

Р. Э. Асратян, канд. техн. наук, вед. науч. сотр, e-mail: rea@ipu.ru,
Институт проблем управления им. В. А. Трапезникова РАН, Москва

Коммуникационно-ориентированная архитектура распределенных информационных систем на основе службы обработки защищенных сообщений

Рассматриваются принципы организации архитектуры распределенных информационных систем, в которой функции прикладных компонентов отделены от функций коммуникационной среды по организации сетевого взаимодействия и информационной защиты (коммуникационно-ориентированной архитектуры). Описывается подход к построению такой архитектуры на основе новой сетевой службы, ориентированной на обработку защищенных сетевых сообщений.

Ключевые слова: *распределенные системы, Web-технологии, интернет-технологии, информационное взаимодействие, информационная безопасность*

Введение

Одна из самых важных тенденций развития сетевых технологий во все последние годы заключалась в постоянном повышении уровня и удобства прикладного программного интерфейса (API) и все большей "изоляции" прикладного пользователя от базисных сетевых понятий. Если еще лет 10—15 назад программирование сетевого взаимодействия на уровне TCP или UDP (т.е. в терминах системных вызовов connect, listen, select, accept и т.п. [1, 2]) применялось не столь уж редко, то сегодня оно уже практически не используется в прикладных разработках. Важную роль в продвижении этой тенденции сыграло создание сетевой архитектуры .NET и, в частности, технологии Web-сервисов. Благодаря поддержке в целом ряде современных систем программирования (MS Visual Studio, Borland C++ builder, Oracle developer suit и т.п.) и Web-серверов (IIS, Apache, Oracle HTTP server), а также удачным сетевым стандартам WSDL и SOAP этой технологии удалось быстро занять лидирующее место в создании сетевых приложений [3, 4]. Наверное, более всех сумели оценить ее достоинства разработчики распределенных информационных систем (РИС), которые как никогда прежде смогли абстрагироваться от подробностей сетевого взаимодействия и сосредоточиться на содержательной стороне проектов.

Тем не менее разработчики РИС сталкиваются и сегодня с рядом проблем. Это связано с тем, что технология Web-сервисов не предлагает готовых решений в такой важной области, как защита и аутентификация данных в сети или же авторизация информационных запросов и разграничение прав доступа к сервисным компонентам (пожалуй, единственным готовым решением является использование защищенного протокола HTTPS в качестве "транспорта" вместе со всеми связанными с ним ограничениями: только одна электронная подпись в данных, невозможность сохранить информационный запрос или ответ вместе с удостоверяющими подписями и т.п.). Кроме того, технология Web-серверов вводит ряд собственных сетевых понятий и конструкций, таких как proxy-class, binding, contract endpoint и т.п., предназначенных для использования в клиентских и серверных компонентах РИС. Все это приводит к порочной практике "перемешивания" прикладных функций компонентов РИС с функциями коммуникационной среды и средств защиты, а в конечном итоге — к увеличению сроков разработки и снижению

уровня информационной безопасности [5, 6]. Важно подчеркнуть, что и сами Web-сервисы являются сетевыми программными модулями особого вида с собственными правилами разработки, отладки и конфигурирования.

В данной работе рассматривается подход к созданию достаточно простой сетевой архитектуры, свободной от отмеченных недостатков.NET. Другими словами, речь идет об архитектуре, ориентированной на защищенную обработку информационных запросов в РИС без использования специфических сетевых конструкций в исходном коде как клиентских, так и сервисных модулей. В целом, предлагаемый подход основан на осознании того, что сетевые средства взаимодействия следует рассматривать не просто как второстепенные по важности добавки к функциональным компонентам РИС, но как "несущую конструкцию", поддерживающую работу всей системы. Здесь можно провести аналогию с приборным шкафом (коммутационная среда), в который с помощью стандартного разъема вставляются аппаратные функциональные модули, не содержащие собственных коммуникационных средств. Сетевую архитектуру РИС, в которой функции прикладных компонент отделены от функций коммуникационной среды по организации сетевого взаимодействия и информационной защиты, будем называть коммуникационно-ориентированной архитектурой (КОА).

Предлагаемый метод организации КОА основан на двух основных понятиях (программных классах): понятии защищенного сетевого сообщения и понятии динамических библиотек сервисных функций, выполняющих обработку таких сообщений. Класс "Защищенное сообщение" отображает контейнер для хранения электронных документов (информационных запросов или ответов), снабженных одной или несколькими удостоверяющими электронными цифровыми подписями (ЭЦП). Отличительная особенность класса "Защищенное сообщение" заключается в том, что он объединяет в себе функции защиты данных с функциями их обработки. В отличие от технологии Web-сервисов, все сервисные функции имеют жесткую сигнатуру: они получают объект класса "Защищенное сообщение" в качестве параметра и возвращают объект того же класса в качестве результата. Разумеется, на электронный документ, содержащийся в этих объектах, не накладывается никаких ограничений (например, это может быть документ в формате XML, содержащий любое число параметров или реквизитов различного типа). Существенным здесь является то, что организация работы с библиотекой функций, размещенной на удаленном сервере, внешне практически не отличается от организации работы с локальной библиотекой, размещенной на рабочей станции. От клиентской программы требуется лишь подключить библиотеку сервисных функций, подготовить исходящее сообщение, содержащее информационный запрос, осуществить вызов требуемой функции для обработки этого сообщения и получить результат выполнения запроса во входящем сообщении. Практически вся работа по установлению соединения с сервером, шифрованию данных в сети, формированию и проверке ЭЦП у исходящего и входящего сообщений и разграничению прав доступа к сервисным функциям выполняется вне кода клиентской программы и вне кода сервисных библиотек.

Короткий пример сетевого взаимодействия

Описываемый подход к построению КОА основан на использовании сетевой службы PMS (Protected Message Service) [7] в качестве базисного средства организации сетевых взаимодействий. Хотя данное применение PMS потребовало доработки и расширения функциональности как клиентской библиотеки функций (PmsBase.dll), так и сервера PMS, свойства ее главного класса PmsMessage ("Защищенное сообщение") внешне практически не изменились. Доработанная служба PMS по-прежнему опирается непосредственно на TCP/IP и не использует никаких элементов Web-технологии (Web-серверы, HTTP, SOAP, WSDL и т.п.).

Рассмотрим простой пример сетевого взаимодействия в нотации языка C# (рис. 1). Предположим, что в каталоге обрабатывающих библиотек сервера PMS с интернет-именем mysrv.ru имеется динамическая библиотека MyLib.dll, в которой определена сервисная функция GetFileList, получающая в качестве параметра полное имя папки в файловой системе сервера и возвращающая список имен файлов, содержащихся в этой папке.

В первой строке примера определяется переменная Request класса PmsMessage. Эта переменная инициируется значением строки, содержащей полное имя папки. Во второй строке к этой переменной добавляются две ЭЦП, сформированные с помощью функции-члена

```
PmsMessage Request=  new PmsMessage("c:\\windows\\system");
Request.AddSignaturesByNames(new string [ ] {"Иванов", "Петров"});

//---------- Вызов функции GetFileList из удаленной (серверной) библиотеки
PmsSrvLibraries RemLibs = new PmsSrvLibraries("mysrv.ru", "O=Мос*| O=Цент*");
if(RemLibs.ErrMsg != "") Console.WriteLine ("Ошибка: " + RemLibs.ErrMsg);
else
{
  PmsMessage  Reply=Request.Process (RemLibs, "MyLib.GetFilelist","T=Зав*");
  if(Reply != null)
    Console.WriteLine (Reply.GetString());
  else
     Console.WriteLine ("Ошибка: " + RemLibs.ErrMsg);
}
RemLibs.Close();

//---------- Вызов той же функции из локальной библиотеки
PmsSrvLibraries LocLibs = new PmsSrvLibraries("c:\\MyLibs\\bin\\Debug");
if(LocLibs.ErrMsg != "") Console.WriteLine ("Ошибка: " + LocLibs.ErrMsg);
else
{
  PmsMessage  Reply=Request.Process (LocLibs, "MyLib.GetFilelist");
  if(Reply != null)
    Console.WriteLine (Reply.GetString());
  else
     Console.WriteLine ("Ошибка: " + LocLibs.ErrMsg);
}
LocLibs.Close();
```

Рис. 1. Пример вызова сервисной функции

(метода) AddSignaturesByNames, обеспечивающей поиск и выборку сертификатов подписантов непосредственно из хранилища пользователя по именам владельцев сертификатов ("Иванов" и "Петров"). Это означает, что с этими сертификатами (вернее, с содержащимися в них открытыми ключами) обязательно должны быть связаны парные им закрытые ключи, иначе формирование ЭЦП закончится аварийно (из примера намеренно удалены операторы обработки исключений).

Четвертая строка примера содержит определение и инициализацию переменной RemLibs класса PmsSrvLibraries, открывающей доступ к библиотекам сервисных функций. Использованный в примере конструктор класса PmsSrvLibraries содержит два строковых параметра, содержащих имя сервера PMS и контрольные значения реквизитов владельца, на основании которых будет определена валидность сертификата сервера.

Логика конструктора в данном случае включает установление сетевого соединения с сервером PMS и немедленный запрос сертификата сервера для последующего шифрования информационных запросов с помощью содержащегося в нем открытого ключа. Важно подчеркнуть, что полученный сертификат проходит не только обычную проверку на корректность (с использованием "доверенного" сертификата), но и проверку на соответствие контрольным значениям реквизитов владельца. В данном примере клиент требует, чтобы название организации владельца сертификата начиналось со строк "Мос" или "Цент", за которыми может следовать все что угодно. В противном случае в открытый член ErrMsg будет занесено диагностическое сообщение о сфальсифицированном сервере. Как видно из примера, контрольные значения реквизитов задаются на основе ключевого формата стандарта X509 [6], но с возможностью использования символов-заместителей ("*" — любая строка, "?" — любой символ) или регулярных выражений.

Если работа конструктора заканчивается успешно, то выполняется вызов сервисной функции с помощью метода Process класса PmsMessage с занесением результата обработки (ответа) в переменную Reply (строка 8). В качестве параметров метода задаются переменная RemLibs, открывающая доступ к удаленным библиотекам, полное имя сервисной функции в формате "имя_библиотеки.имя_функции" и, опционально, контрольные значения реквизитов подписантов в ответе сервера. В данном примере клиент требует, чтобы название должности подписанта начиналось со строки "Зав". Обработка запроса завершается выводом на консоль или полученного результата (списка имен файлов), извлеченного из переменной Reply, или сообщения об ошибке. Важно подчеркнуть, что дешифрование запроса и формирование ЭЦП для результата выполняются сервером PMS автоматически без участия сервисной функции на основании сертификатов, заданных в его конфигурации. Точно так же шифрование ответа перед отправкой в сеть сервер выполняет автоматически (с использованием сертификатов, извлеченных из ЭЦП, содержащихся в запросе). Дешифрование же ответа и проверка его подписей выполняются на рабочей станции в рамках метода Process и без участия программы клиента.

В нижней половине рис. 1 проиллюстрировано обращение к той же функции, но из локальной библиотеки, размещенной в каталоге "c:\MyLibs\bin\Debug" рабочей станции. Внешне все выглядит почти точно так же, только вместо имени сервера в конструкторе PmsSrvLibraries указано имя каталога, а контрольные значения реквизитов сертификата отсутствуют, так как шифрование данных в таком случае не требуется. Однако логика обработки меняется радикально. Вместо установления соединения с сервером PMS конструктор выполняет поиск всех динамических

библиотек в указанном каталоге и поиск всех функций, принимающих объект класса PmsMessage в качестве параметра и возвращающих объект этого же класса в качестве результата, в каждой библиотеке. Информация обо всех найденных функциях сохраняется в специальной структуре данных класса PmsSrvLibraries, обеспечивающей быстрый поиск функции по полному имени и доступ к ней. Соответственно меняется и логика выполнения метода Process класса PmsMessage: вместо обращения к серверу PMS осуществляется вызов функции локальной библиотеки. Данный способ вызова предназначен для отладки сервисных функций и их взаимодействия с клиентом вне сетевой среды. Если на рабочей станции установлена среда разработки (например, MS Visual Studio) и имеются проекты клиентского модуля и библиотеки сервисных функций, то можно провести отладку функции с использованием всего инструментария отладочной среды, включая точки останова и пошаговое выполнение. Отлаженная библиотека может быть перенесена на сервер PMC (в специальный каталог серверных библиотек, заданный в конфигурации сервера) без каких-либо изменений.

Оператор RemLib.Close() закрывает сетевое соединение с сервером PMS, а оператор LocLib.Close() освобождает локальные ресурсы.

Определение сервисной функции для данного примера в нотации c# приведено на рис. 2. Здесь важно обратить внимание на сигнатуру функции: все сервисные функции независимо от назначения и логики обработки должны иметь одинаковую сигнатуру. Первым и главным параметром функции является объект класса PmsMessage, содержащий информационный запрос. Второй параметр Conf содержит конфигурационные данные в форме массива строк вида "имя = значение". Через третий строковый параметр Msg функция передает диагностическое сообщение (которое становится доступным клиенту через член ErrMsg класса PmsSrvLibraries). Возвращаемое значение функции — объект класса PmsMessage, содержащий результат обработки. Сервисные функции, имеющие иную сигнатуру, сервер игнорирует.

```
public class MyLib
{
    public PmsMessage GetFileList(PmsMessage Inp, string [ ] Conf, ref string Msg)
    {
        string sRes = "";
        try
        {
            string Dir = Inp.GetString();
            string [ ] sFiles = Directory.GetFiles(Dir);
            if (sFiles != null)
            {
                for (int i = 0; i < sFiles.Length; i++)
                  sRes += sFiles[i] + "\r\n";
            }
        }
        catch(Exception Ex)
        {
            Msg=Ex.Message;
            if(PmsBase.Cfg(Conf, "Debug") =="yes")
                  (PmsBase.Log(PmsBase.Cfg(Conf, "Log"), Ex.Message);
            return null;
        }
        Msg = "OK";
        PmsMessage Res = new PmsMessage(sRes);
        return Res;
    }
    . . .
}
```

Рис. 2. Пример определения сервисной функции

При подключении каждой динамической библиотеки сервер PMS выполняет поиск ее конфигурационного файла, совпадающего по имени с файлом библиотеки, но имеющего расширение имени ".cfg" (своего рода аналог файла web.config для Web-сервисов). Этот файл может содержать определения строковых конфигурационных параметров в форме "имя = значение" как для библиотеки в целом, так и для каждой функции индивидуально. Конфигурационные данные сохраняются во внутренних структурах данных сервера и автоматически подаются в качестве значения параметра Conf на вход каждой функции при обращении к ней. Пример конфигурационного файла библиотеки приведен на рис. 3. Отметим, что индивидуальные параметры функции размещаются после заголовка вида "[имя_функции]". Разрешается вводить параметры с любыми именами, но некоторые параметры имеют системное значение и интерпретируются сервером PMS. К последним относятся, например, параметры Timeout, определяющий тайм-аут времени выполнения функции в секундах; Access, обеспечивающий разграничение прав доступа к ней. Подчеркнем, что тайм-аут мож-

```
Log=c:\PmsLog\MyLib.log
. . .
[GetFileList]
Timeout=5
Access=T=Зав*,CN=Петров * | T=Зам*,CN=Сидоров *
Debug=yes

[GetFile]
Access=T=Директор*
. . .
```

Рис. 3. Пример конфигурационного файла библиотеки сервисных функций

но определять для каждой функции в отдельности (в отличие от Web-сервисов). Параметр Access задает набор контрольных значений реквизитов подписантов информационного запроса в уже знакомой нам нотации, на основе которого сервер PMS ограничивает доступ к той или иной функции. Например, доступ к функции GetFileList (см. рис. 3) открыт только тем запросам, среди подписантов которых имеется владелец сертификата с названием должности, начинающимся со строки "Зав", и фамилией "Петров" или же с названием должности, начинающимся со строки "Зам", и фамилией "Сидоров".

Как видно из рис. 2, функция GetFileList извлекает имя папки из параметра Inp и формирует список имен файлов, содержащихся в этой папке, в форме строки, используя статический метод GetFiles класса Directory. Полученный список помещается в новый объект класса PmsMessage и возвращается клиенту.

При возникновении исключительной ситуации (например, если указанная папка не существует) функция считывает значение конфигурационного параметра "Debug" из массива Conf с помощью статического метода PmsBase.Cfg, определенного в клиентской библиотеке PmsBase.dll. Если этот параметр имеет значение "yes", функция создает запись в файле журнала, определенном конфигурационным параметром "Log", с помощью статического метода PmsBase.Log.

Протокол PMS

PMS в полной мере использует двоичную природу TCP/IP [7, 8]. Взаимодействие между клиентом и сервером PMS осуществляется по специальному, достаточно простому PMS-протоколу, ориентированному на передачу двоичных сетевых сообщений (PMS-сообщений) в обоих направлениях (никакие преобразования двоичных данных в текстовую форму типа base64 не применяются). Каждое такое сообщение в общем случае содержит два массива байтов: заголовок сообщения и тело сообщения (рис. 4). Первые 4 байта заголовка или тела сообщения содержат целое число — его длину. При передаче запроса от клиента к серверу в заголовок сетевого сообщения помещается строка, содержащая полное имя вызываемой функции, а в тело сообщения упаковывается структура PmsMessage в открытой или зашифрованной форме, содержащая информационный запрос.

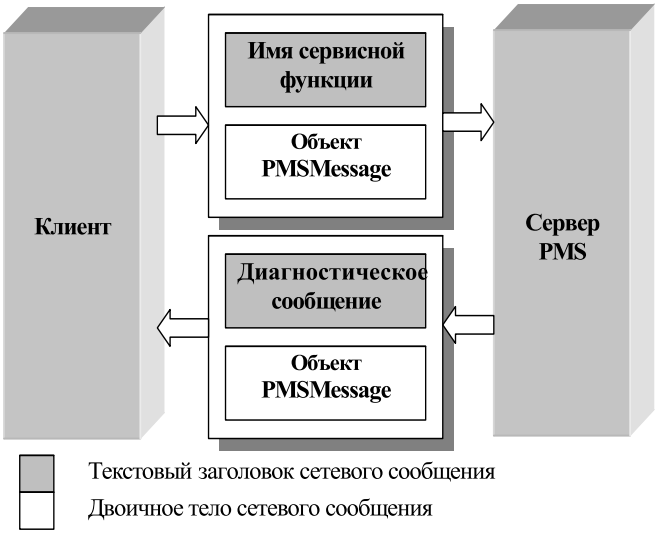

Рис. 4. Сетевые сообщения

Строку заголовка сервер использует для организации вызова соответствующей обрабатывающей функции. При передаче результата обработки от сервера к клиенту в заголовок сетевого сообщения помещается строка диагностического сообщения (значение параметра Msg, сформированное обрабатывающей функцией), а в тело сообщения упаковывается структура PmsMessage, содержащая ответ сервера в открытой или зашифрованной форме, предварительно подписанный собственным закрытым ключом сервера. Никакие двоично-текстовые преобразования (типа base64) не применяются. Полученное от сервера диагностическое сообщение автоматически присваивается члену ErrMsg объекта класса PmsSrvLibraries на стороне клиента (см. рис. 1).

Классы PmsMessage и PmsSrvLibraries

Главная особенность класса PmsMessage заключается в том, что он содержит всего два элемента данных (поля):

- открытый (public) массив байтов Data, предназначенный для хранения передаваемых данных (т.е. "полезной нагрузки");
- массив электронных подписей для передаваемых данных, каждая из которых содержит не только собственно подпись ("свертку" данных, зашифрованную закрытым ключом подписанта), но и удостоверяющий сертификат, включающий парный открытый ключ и основные реквизиты подписанта (страна, город, организация, должность и т.п.).

Хотя класс PmsMessage включает довольно много функций-членов (для формирования,

проверки и удаления ЭЦП, для шифрования/дешифрования данных и для передачи их на обработку сервисным функциям), он содержит всего два конструктора: PmsMessage(byte [] Bytes) и PmsMessage(string Str), позволяющих инициировать член Data данными из заданного массива данных или из символьной строки соответственно. Если пользователю нужно передавать объекты сложной структуры, то он должен самостоятельно запрограммировать средства "упаковки" таких объектов в массив байтов на отправляющей стороне и для "распаковки" на принимающей стороне (или же воспользоваться стандартными средствами байтовой сериализации).

Чтобы упростить задачу пользователя, в библиотеке PmsBase.dll определен класс PmsDocument, который можно рассматривать

```
class PmsDocument
{
   string Doc;                          //текстовый документ
   Attachment [] AttArray;      //массив приложений
   // конструктор («распаковка»)
   PmsDocument (PmsMessage Pmsg) {
     . . .
   }
   // создание защищенного сообщения («упаковка»)
   PmsMessage GetPmsMessage() {
     . . .
   }
}

class Attachment
{
   string Name;            //имя файла приложения
   string Comment;     //дополнительная информация
   byte [] Data;          //данные приложения
}
```

Рис. 5. Определение класса PmsDocument

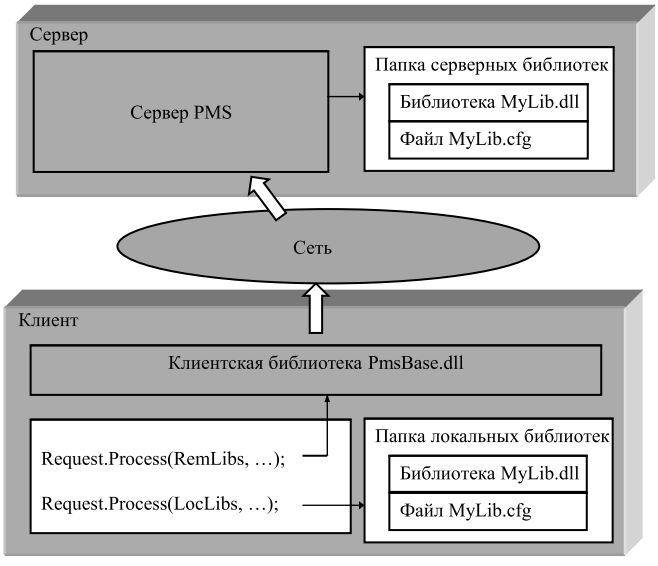

Рис. 6. Работа с удаленными и локальными библиотеками

как готовый пример программной "надстройки" над классом PmsMessage. Этот класс предназначен для передачи информационных запросов и информационных ответов в РИС и содержит символьную строку для хранения текстового документа (например, в формате XML) и массив двоичных приложений к этому документу (графической информации, документов в формате PDF, MS-Word и т.п.). Определение класса на языке C# проиллюстрировано на рис. 5.

Алгоритмы "упаковки" объекта PmsDocument в массив байтов и "распаковки" в первоначальный вид (метод GetPmsMessage и конструктор PmsDocument) мы рассматривать не будем.

Логика работы с библиотеками сервисных функций, заложенная в класс PmsSrvLiblaries, проиллюстрирована на рис. 6. Имена переменных Request, RemLibs и LocLibs в нижней части этого рисунка имеют то же смысл, что и на рис. 1.

Как видно из рис. 6, обращение к удаленной библиотеке сервисных функций через переменную RemLib выливается в передачу информационного запроса серверу PMS, который заранее (при запуске) просматривает свою папку серверных библиотек и собирает сведения обо всех найденных динамических библиотеках, сервисных функциях и их конфигурационных параметрах в своей внутренней структуре данных для обеспечения быстрого доступа к ним.

Обращение к локальной библиотеке сервисных функций через переменную LocLib выливается в прямой вызов функции из локальной динамической библиотеки без участия сервера PMS. Сбор сведений обо всех найденных динамических библиотеках, сервисных функциях и их конфигурационных параметрах выполняет конструктор PmsSrvLibraries при инициализации переменной LocLibs. Важно отметить, что в этом случае программа клиента и сервисная функция выполняются не только на одной машине, но и в контексте одного процесса, что очень удобно для отладки их взаимодействия.

Несколько слов о сетевых настройках. Все сетевые технологии, включая и Web-сервисы, допускают задание значений разнообразных сетевых настроек (размер буфера чтения, размер буфера записи, тайм-ауты ожидания и т.п.), но одновременно предлагают набор разумных значений по умолчанию, с тем чтобы пользователю почти никогда не пришлось иметь с ними дело. Как правило, эта цель достигается в отношении большинства настро-

ек, кроме одной: тайм-аута чтения данных из сети. Разработчикам РИС часто приходится "напрямую" управлять значением этого параметра, так как оно должно быть согласовано со временем обработки информационного запроса: если в клиентской программе тайм-аут чтения равен 120 с, а время обработки запроса на сервере может достигать 300 с, то пользователь рискует никогда не получить результата.

Конструктор PmsSrvLibraries также допускает задание сетевых настроек через дополнительные параметры, если их значения по умолчанию не удовлетворяют пользователя. Однако главная особенность PMS в части управления сетевыми настройками связана со значением тайм-аута чтения из сети в клиентской программе — это значение она принимает от сервера. Как уже отмечалось при обсуждении рис. 1, работа конструктора PmsSrvLibraries при подключении удаленных библиотек начинается с установления соединения с сервером и с запроса сертификата сервера. Предложенное решение основано на том, что ответ на этот запрос содержит не только сертификат, но и еще одну важную характеристику — максимальное значение тайм-аута выполнения сервисной функции, заданное в конфигурационных файлах серверных библиотек (конфигурационный параметр Timeout), увеличенное на 10 с. Если это значение больше клиентского тайм-аута чтения, то сетевое соединение с сервером закрывается, устанавливается новое значение тайм-аута чтения, равное полученному от сервера значению, после чего соединение с сервером немедленно восстанавливается, но уже с новой настройкой. Такое решение гарантирует, что клиент обязательно дождется результата выполнения сервисной функции или же диагностического сообщения сервера о прерывании ее выполнения вследствие тайм-аута.

Временные оценки

Для проведения лабораторных экспериментов и получения сравнительных оценок быстродействия была проведена реализация службы PMS в среде программирования Microsoft VisualStudio 2010 для платформы.Net Framework 4.0 на основе криптосистемы "КриптоПро CSP" версии 3.6, соответствующей требованиям действующих в России ГОСТов в области криптографической защиты информации.

Основная цель экспериментов с PMS заключалась в сравнении ее быстродействия с быстродействием Web-сервисов в одинаковых условиях. Главное внимание уделялось вызовам сервисных функций с относительно малым (от нескольких миллисекунд до нескольких сотен миллисекунд) временем выполнения (при более длительной обработке разница между двумя технологиями практически нивелируется) с применением средств криптозащиты и без применения. В экспериментах с PMS использовали средства криптозащиты, интегрированные в клиентскую библиотеку и сервер PMS. В экспериментах с Web-сервисами средства криптосистемы "КриптоПро CSP" подключались непосредственно к программе клиента и программе Web-сервиса. При этом сервер PMS с модельными библиотечными функциями и Internet Information Server с модельными Web-сервисами были установлены на одном и том же четырехъядерном сервере приложений с тактовой частотой 2,4 ГГц в операционной среде Window 2003 Server. В качестве клиентской рабочей станции был использован одноядерный компьютер с тактовой частотой 2,8 ГГц.

На рис. 7—9 показаны характерные результаты экспериментов с очень быстрой сервисной функцией, выполняющей простое перекодирование полученного строчного сообщения в верхний регистр и возврат результата клиенту, при длине сообщения в 2, 50 и 100 Кбайт соответственно. На каждом рисунке приведены диаграммы времени выполнения операции на сервере с помощью Web-сервиса (черный столбик) и с помощью PMS (серый столбик) для четырех режимов: без применения криптозащиты; с применением ЭЦП; с применением ЭЦП и шифрования сообщений и, наконец, с применением ЭЦП, шифрования и проверки сертификатов на корректность. Проверку сертификатов выполняли путем построения "цепочки" сертификатов от проверяемого сертификата до "доверенного". В каждом режиме время выполнения вычисляли, как среднее значение для 100 последовательных вызовов сервисной функции.

Как видно из рис. 7—9, в данной серии экспериментов PMS не уступает технологии Web-сервисов в быстродействии и даже несколько превосходит ее. Причем в режиме "без криптозащиты" это превосходство является весьма значительным (что, по-видимому, объясняется временными затратами, связанными с применением протокола HTTP/SOAP). При подключении криптозащиты время обработки возрастает и различие становится менее существен-

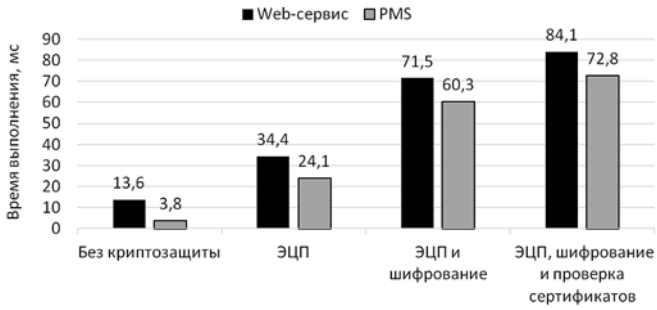

Рис. 7. Время обработки при длине сообщения 2 Кбайт

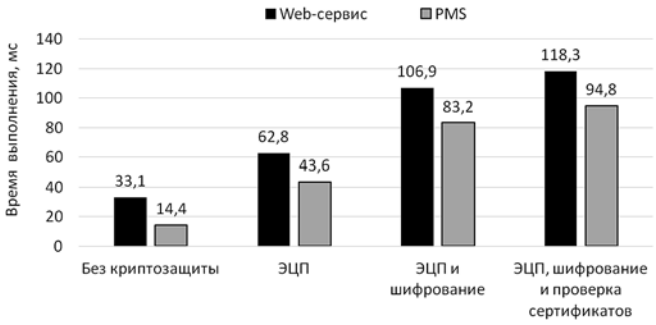

Рис. 8. Время обработки при длине сообщения 50 Кбайт

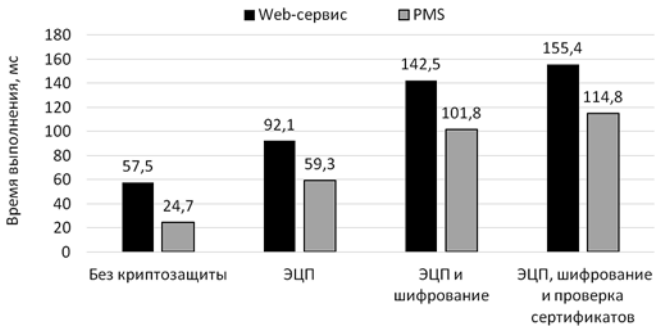

Рис. 9. Время обработки при длине сообщения 100 Кбайт

ным (характерно, что абсолютная разница во времени выполнения при этом остается относительно стабильной).

Заключение

Фактически предложенный подход к построению КОА основан на отказе от гибкой модели организации взаимодействия, основанной на вызове методов удаленных объектов (например, через Web-сервисы), в пользу более жесткой модели, основанной на обмене защищенными сообщениями. Введение стандартной сигнатуры сервисных функций (получают объект класса "Защищенное сообщение" в качестве главного параметра и возвращают объект этого же класса в качестве результата) не представляется слишком уж ограничительным и неудобным, когда речь идет об обработке информационных запросов в распределенных системах, так как оставляет возможность обмена электронными документами любой сложности. Вместе с тем эта унификация позволяет значительно продвинуться в направлении создания готовых решений таких важных для разработчика задач, как защита данных в сети, аутентификация и авторизация информационных запросов, разграничение прав доступа к сервисным функциям или же отладка клиентских и серверных компонентов вне сетевой среды. Основное же преимущество данного подхода заключается в том, что он позволяет сделать шаг в направлении главного требования КОА — разгрузке кода клиентских программ и библиотек сервисных функций от специфических программных конструкций, направленных на организацию безопасных взаимодействий в сети.

Если продолжить аналогию КОА с приборным шкафом, которая была использована во введении, то в описанном подходе классам PmsMessage и PmsSrvLibraries, по-видимому, выпадает роль "стандартного разъема" для подключения клиентских модулей в РИС (см. рис. 1), а жесткой сигнатуре сервисных функций — роль "стандартного разъема" для подключения серверных компонентов (см. рис. 2).

Список литературы

1. **Снейдер Й.** Эффективное программирование TCP/IP. Библиотека программиста. СПб.: Символ-Плюс, 2002. 320 с.

2. **Хант К.** TCP/IP. Сетевое администрирование. — СПб.: Питер, 2007. 816 с.

3. **Шапошников И. В.** Web-сервисы Microsoft.NET. СПб: БХВ-Петербург, 2002. 336 с.

4. **Мак-Дональд М., Шпушта М.** Microsoft ASP.NET 3.5 с примерами на C# 2008 и Silverlight 2 для профессионалов. М.: Вильямс, 2009. 1408 с.

5. **Згоба А. И., Маркелов Д. В., Смирнов П. И.** Кибербезопасность: угрозы, вызовы, решения // Вопросы кибербезопасности. 2014. № 5. С. 30—38.

6. **Козлов А. Д., Орлов В. Л.** Методы и средства обеспечения информационной безопасности распределенных корпоративных систем. — М.: ИПУ РАН, 2017. — 156 с.

7. **Асратян Р. Э.** Интернет-служба защищенной обработки информационных запросов в распределенных системах // Программная инженерия. 2016. № 11. С. 490—497.

R. E. Asratian, Leading Researcher, rea@ipu.ru,
V. A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences,
Moscow, 117997, Russian Federation

Communication-Oriented Architecture of Distributed Information Systems on the Base of Protected Message Service

The principles of the organization of architecture of the distributed information systems in which functions of application-oriented components are separated from functions of the communication environment ("communication oriented" architecture) are considered. An approach to creation of such architecture on the basis of the new network service oriented on protected processing of the network messages (Protected Message Service — PMS) is describe.Distinctive feature of service is the close integration of authentication and data protection functions with functions of network information exchange. From the client point of view the service architecture is based on two main program classes: "Protected message" and "Remote Library Set". This classes offer necessary functionality not only for creating and protecting messages, but also for transferring them to remote server via established network connections for processing. Contrary to web-cervices, based on remote function call model, PMS-service is based on message processing model: all service functions receive object of "Protected message" class as a parameter and return another object of the same class as a result of processing. Main advantages of proposed service include high speed of processing (small overhead for data transmission) and a possibility of debugging both client, and server components of system out of the network environment. The experimental implementation of PMS in C# for Microsoft Framework 4.0 and the study of performance of new service (in comparison with web services in.NET architecture) were carried out and the area of its effective application is outlined. This area includes distributed systems in which requirements of high processing rate and information security are viewed more important, than flexibility in service functions specifications.

Keywords: *distributed systems, Web-technologies, Internet-technologies, network interactions, data security.*

References

1. **Snader J.** *Effektivnoe programmirovanie TCP/IP* (Effective TCP/IP programming), SPb.: Simvol-Pljus, 2002. 320 p. (in Russian).

2. **Hunt C.** *TCP/IP. Setevoe administrirovanie* (TCP/IP Network administration), SPb.: Piter, 2007, 816 p. (in Russian).

3. **Shaposhnikov I. V.** *Web-servisy Microsoft.NET* (Web-services of Microsoft.NET), SPb: BHV-Peterburg (), 2002. 336 p. (in Russian).

4. **MacDonald M., Szpuszta M.** Microsoft *ASP.NET 3.5 s primerami na C# 2008 i Silverlight 2 dlja professionalov* (Pro Microsoft ASP.NET 3.5 in C# 2008 includes Silverlight 2), Moscow, Viljams, 2009. 1408 p. (in Russian).

5. **Zgoba A. I., Markelov D. V., Smirnov P. I.** Kiberbezopasnost: ugrozy, vyzovy, reshenija (Cybersafety: threats challenges, decisions), *Voprosy kiberbezopasnosti*, 2014, no. 5, pp. 30—38 (in Russian).

6. **Kozlov A. D., Orlov V. L.** *Metody i sredstva obespechenija informacionnoj bezopasnosti raspredelennyh korporativnyh sistem* (Methods and means of ensuring of information security in distributed enterprise systems), Moscow, IPU RAN, 2017, 156 p. (in Russian).

7. **Asratian R. E.** Internet-sluzhba zashhishhennoj obrabotki informacionnyh zaprosov v raspredelennyh sistemah (Internet service for protected information queries processing in distributed systems), *Programmnaya inzheneria*, 2016, no. 11, pp. 490—497.