macromedia PRESS

КУДИЦ-ОБРАЗ

ПОПУЛЯРНЫЕ WEB-ПРИЛОЖЕНИЯ

на FLASH MX

ИГРЫ
ГОСТЕВЫЕ КНИГИ
ЧАТЫ
ФОРУМЫ
ДОСКИ ОБЪЯВЛЕНИЙ
ГОЛОСОВАНИЯ
и многое другое

Win/Mac/Unix
CD-ROM в комплекте

Тим К. Чанг Шон Кларк Эрик Е. Долецки Джон Игнацио Джелос
Михаэль Грюндвиг Д кар Макс Ошман Вильям Б. Сандерс Скотт Смит

ИЗДАТЕЛЬСТВО
«КУДИЦ-ОБРАЗ»

macromedia®

FLASH MX

CREATING DYNAMIC APPLICATIONS

INTEGRATED TECHNOLOGIES

Macromedia Flash™

Java™

ColdFusion®

.NET

Tim K. Chung • Sean Clark • Eric E. Dolecki • Juan Ignacio Gelos • Michael Grundvig
Jobe Makar • Max Oshman • Dr. William B. Sanders • Scott Smith

Тим К. Чанг • Шон Кларк • Эрик Е. Долецки • Джон Игнацио Джелос
Михаэль Грюндвиг • Джоб Макар • Макс Ошман • Вильям Б. Сандерс • Скотт Смит

ПОПУЛЯРНЫЕ WEB-ПРИЛОЖЕНИЯ НА FLASH MX

Перевод с английского

КУДИЦ-ОБРАЗ
МОСКВА • 2003

Чанг Т. К., Кларк Ш. и др.
Популярные web-приложения на FLASH MX. Пер. с англ. - М.: КУДИЦ-ОБРАЗ, 2003 - 272 с.

Данная книга является хорошим руководством по практической стороне разработки динамических приложений в среде Flash MX. Авторы последовательно излагают принципы программирования с использованием языка ActionScript. Каждая глава посвящена разработке какого-нибудь законченного приложения, будь то клиент электронной почты или система обмена мгновенными сообщениями. Исходный код всех примеров подробно описан и тщательно разбирается. Серверная часть представлена платформами Java, .NET и ColdFusion. Изучив представленный здесь материал, вы сможете самостоятельно разрабатывать Flash-приложения любого типа. Если ваш девиз - "практика, практика и еще раз практика", то эта книга для вас...

ISBN 0-321-11548-1
ISBN 5-93378-079-0

Тим К. Чанг, Шон Кларк, Эрик Е. Долецки, Джон Игнацио Джелос, Михаэль Грюндвиг, Джоб Макар, Макс Ошман, Вильям Б. Сандерс, Скотт Смит.
Популярные web-приложения на FLASH MX.

Учебно-справочное издание

Корректор М. Матекин
Перевод с англ. Е. Смогайлов
Научный редактор И. Кошечкин

Лицензия ЛР № 071806 от 02.03.99. НОУ «ОЦ КУДИЦ-ОБРАЗ».
119034, Москва, Гагаринский пер., д. 21, стр. 1. Тел.: 333-82-11, ok@kudits.ru

Подписано в печать 26.08.2003.
Формат 70x100/16. Печать офсетная.
Усл. печ. л. 21,9. Тираж 3000 экз. Заказ № 519.

Отпечатано с готовых диапозитивов в ООО «Типография ИПО профсоюзов Профиздат».
109044, Москва, Крутицкий вал, 18.

ISBN 0-321-11548-1
ISBN 5-93378-079-0

1. Динамическая программа опросов

Автор Макс Ошман (Max Oshman)

Вы, наверно, много раз задумывались о том, что именно думают люди о вашем Интернет-сайте и каким образом увеличить его посещаемость. Существует много возможных решений; однако только одно из них совмещает простоту с удобством использования вашими посетителями: проведение опроса[1].

У всех есть свое мнение, и каждый с удовольствием им поделится. Что может быть лучше чем опрос для выяснения чувств ваших посетителей?

В этой главе мы всесторонне изучим программу опросов, написанную на ColdFusion, ASP.NET и Java.

В качестве клиента (front-end), несомненно, используется Flash, что дает возможность продемонстрировать многие из новых возможностей среды MX.

Перед тем как продолжить чтение, рекомендуется изучить Flash, ColdFusion и XML на среднем или более высоком уровне, так как в противном случае вы не сможете понять большую часть идей и технических деталей, используемых при написании этой программы.

Что это такое?

Опрос является превосходным инструментом для сбора информации от посетителей. Пользователи часто с большой опаской относятся к анкетам, так как на их заполнение требуется некоторое время, а в наше время быстрых Интернет-соединений и обедов на заказ мало кто склонен тратить свое скудное свободное время на заполнение анкет. Опрос очень хорошо подходит для того, чтобы быстро получить ответы на интересующие вас вопросы, например такие, как "Что бы вы хотели увидеть в ближайшее время на нашем сайте?" или "Нравится ли вам новый вид нашего сайта?". Вопросы и ответы можно легко обновить, а результаты доступны без всяких усилий. В конечном итоге опрос является легким и эффективным путем получения информации от посетителей вашего сайта.

1. В нашей стране более широко используется термин "система голосований". - *Примеч. науч. ред.*

Каким образом это будет работать?

Всем нам необходим некоторый порядок в жизни (поэтому нам так нравится XML), следовательно, перед началом любого проекта необходимо понять, что именно нужно сделать и каким образом. Многие любят рисовать и начинают с блок-схем, конкретный способ действий уже зависит от вас, при условии что у вас есть план. В нашем случае опрос состоит из admin (конфигурационной программы), написанной на HTML, которая позволяет изменить вопрос и набор возможных ответов. Каждый вопрос может иметь неограниченное число ответов. Пользователь может проголосовать не более одного раза по каждому вопросу. Только один вопрос может быть активен в какой-то момент времени, с неограниченным числом ответов. Ниже описано, как Flash-клиент будет взаимодействовать с сервером.

Порядок работы с сервером со стороны клиента

1. Flash посылает серверу запрос на данные и результаты текущего опроса.
2. Flash спрашивает у сервера, голосовал ли данный пользователь в текущем опросе.
3. Если пользователь еще не голосовал, то Flash показывает данные опроса. В противном случае выводятся результаты опроса.

Детали взаимодействия клиента с сервером

1. Flash получает вопрос и список ответов.
   а) Посылается запрос "GetPollData" на транзакцию.
   б) Сервер передает вопрос, ответы, а также текущие результаты.
2. Flash спрашивает у сервера, голосовал ли раньше данный пользователь.
   а) Посылается запрос "HasVoted" на транзакцию.
   б) Сервер возвращает истинное или ложное значение.
3. Flash посылает результат голосования.
   а) Посылается на транзакцию запрос "HasVoted", содержащий PollID (идентификатор опроса), ассоциированный с AnswerID (идентификатор ответа).
   б) Сервер отвечает сообщением об успехе или ошибке.

Для передачи данных между Flash-клиентом и сервером используется XML. Все данные (вопросы, ответы, результаты и т. д.) хранятся в базе данных Access.

База данных

База данных является основой нашей программы. В ней хранятся вопросы, ответы и результаты опросов. Так как наш проект довольно мал, в качестве базы данных выбрана Microsoft Access. Если ваш проект требует хранения очень больших объемов данных, рекомендуется использовать Oracle (в настоящий момент версия 9i).

Для того чтобы лучше понять, как работает опрос, давайте взглянем на ключевые моменты хранения данных.

Откройте файл Poll-97.mdb с CD-ROM-диска. На рис. 1.1 показана законченная база данных. Вы увидите две таблицы: Answers (ответы) и Polls (опросы). Каждая таблица имеет свое значение; вместе они обеспечивают правильную работу опроса.

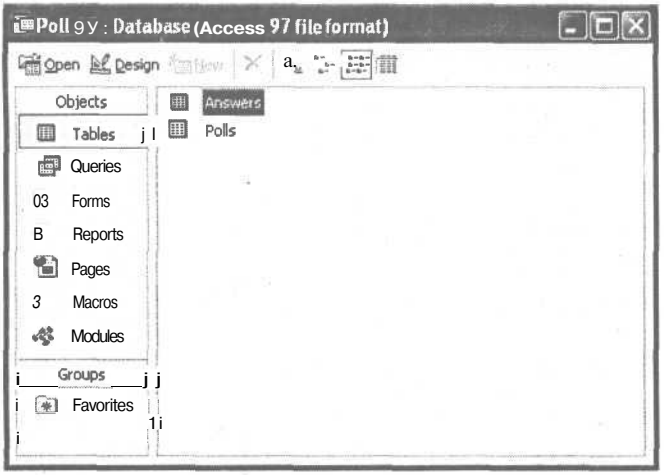

Рис. 1.1. База данных, необходимая для работы опроса, состоит из двух таблиц

Ответы

В таблице Answers хранятся ответы всех опросов (ответы вносятся в базу данных с помощью программы admin, при добавлении вопроса). В этой таблице также хранится число голосов, поданных за каждый ответ.

Опросы

В таблице Polls хранятся все вопросы, при этом для каждого вопроса хранится: время создания, активен ли вопрос в данный момент, а также общее число голосов, полученных на данный вопрос. Вопрос добавляется с помощью программы admin, так же как и ответы.

XML-документ

Как уже было сказано раньше, для обмена данными между Flash-клиентом и сервером используется документ XML. Фрагменты данного XML-документа появляются в разных местах нашей программы, с целью организации информации, посылаемой в базу данных или извлекаемой из базы данных. Когда мы будем обсуждать код, написанный на ColdFusion, Java и ASP.NET, большая часть XML вам будет уже знакома.

Откройте файл Poll.xml с CD-ROM- диска.

Ниже приведен XML-документ, используемый во всех трех вариантах опроса (написанных на ColdFusion, ASP.NET и Java).

```
<Blah>

<!- Запрос на получение вопроса и ответов опроса ->
<Request>
      <TransactionType>GetPollData</TransactionType>
      <Data />
</Request>

<!- Ответ на транзакцию GetPollData ->
<Response>
   <Status>Success</Status>
   <Data>
      <Poll ID="" TotalVotes="">
         <Question ID="">
            <Text></Text>
            <Answers>
               <Answer ID="" Votes="">
                  <Text></Text>
               </Answer>
               <Answer ID="" Votes="">
                  <Text></Text>
               </Answer>
            </Answers>
         </Question>
      </Poll>
   </Data>
</Response>

<!- Запрос на голосование ->
<Request>
   <TransactionType>VoteOnPoll</TransactionType>
   <Data>
      <Vote PollID="" QuestionID="" AnswerID="" />
   </Data>
</Request>

<!– Ответ на предыдущий запрос ->
<Response>
   <Status>Success</Status>
   <Data>
      <Message>Your vote was recorded</Message>
   </Data>
</Response>

<!- Запрос о том, голосовал ли раньше пользователь в текущем опросе ->
```

```
<Request>
   <TransactionType>HasVoted</TransactionType>
   <Data></Data>
</Request>

<!- Ответ на HasVoted транзакцию ->
<Response>
   <Status>Success</Status>
   <Data>
      <VotedAlready>True | False</VotedAlready>
   </Data>
</Response>

<!- Ответ в случае возникновения ошибки ->
<Response>
   <Status>Error</Status>
   <Data>
      <Message>An error has occurred!</Message>
   </Data>
</Response>
</Blah>
```

Конфигурационная программа (admin)

Программа admin используется для обновления вопросов и ответов опроса. Хотя можно также поменять вопрос и ответы вручную, это довольно утомительное занятие. С помощью admin весь опрос можно поменять за несколько минут. Вдобавок, если у вас появится в будущем гораздо более обширное приложение с требованием быстрого обновления данных, вам очень пригодятся использованные здесь методы.

Откройте файл admin.cfm с CD-ROM-диска. Законченная программа admin показана на рис. 1.2.

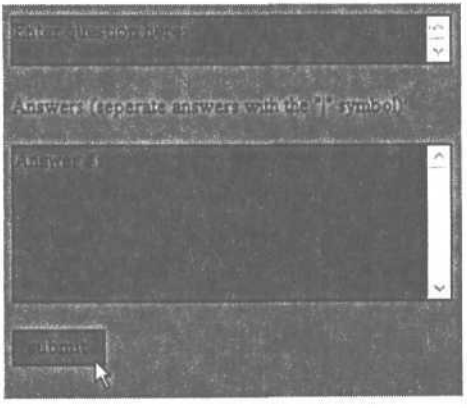

Рис. 1.2. Обновление данных опроса с помощью admin

Код программы admin приведен ниже.

```
<html>
<head>
<Title>Poll Admin</title>

<!-- Внешний вид формы определяется с помощью CSS (каскадных таблиц стилей) -->
<style type="text/css">

TEXTAREA, INPUT {
    font-family: Georgia, "MS Serif, "New York", serif;
    padding: 1px;
    font-size: 10px;
    color: #000000;
    background-color: #666666;
    border: solid 1px #000000;
}
</style>
</head>

<body bgcolor="#999999">

<cfparam name="action" default="none">
<cfswitch expression="#action#">
    <cfcase value="none">

        <form name="info" method='post">

    <p>
    <textarea name="question" cols="50" class="texta1"
    Ж wrap="VIRTUAL">Enter question here:</textarea>
    <p> Answers (separate answers with the "|" symbol):
    <p>

        <textarea name="answers" cols="50" class="texta1"
        Ж wrap="VIRTUAL" rows="7">Answer #1 </textarea>
<p>
        <|nput type="submit" value="submit">
        <input type="hidden" name="action" value="insert">
        </form>

    </cfcase>

    <cfcase value="insert">

        <!-- Сохраняем в таблице опроса (Poll) вопрос и дату его создания -->
        <cfquery name="InsertQuestion" datasource="POLL" dbtype="odbc">
            insert into Polls
            (pollquestion, createdate)
            values ('#form.question#', #CreateODBCDate(Now())#)
        </cfquery>

        <!-- Выборка идентификатора опроса(PollID) с наибольшим значением и присвоение
        Ж этого значения переменной HighID-->
        <cfquery name="GetNewID" datasource="POLL" dbtype="ODBC">
            Select Max(PollID) as HighID from Polls
        </cfquery>
```

```
<!-- Делаем неактивными все вопросы -->
<cfquery name="deactivate" datasource="poll">
    UPDATE Polls
    set active = 0
</cfquery>

<!-- Активизируем опрос с идентификатором (PolllD), равным HighID
Ж (активизируется опрос, добавленный последним) -->
<cfquery name="active" datasource="poll">
    UPDATE Polls
    set active = 1
    where PolllD=#GetNewID.HighID#
</cfquery>

<!-- Цикл по всем ответам для добавления их в базу данных по одному -->
<cfloop list="#form.answers#" index="list_index" delimiters="|">

<cfquery name="InsertAnswer" datasource="POLL" dbtype="odbc">
    insert into Answers
        (polllD, answer)
    values (#GetNewID.HighID#,'#list_index#')
</cfquery>

</cfloop>

<!- Сообщение пользователю об успешном обновлении опроса ->
<cfoutput>
    <br>The poll has been updated
</cfoutput>

    </cfcase>
</cfswitch>

</body>
</html>
```

Примечание: *Весь HTML-код должен быть вам понятен, может быть за исключением кода внутри <style>-элементов. Это код для CSS (каскадных таблиц стилей). CSS позволяет контролировать раскладку ваших HTML-документов с помощью простого кода пригодного для многократного использования. Дополнительную информацию о CSS можно получить на сайте www.w3.org/Style/CSS/.*

Часть кода, написанная на ColdFusion, также довольно проста. Сначала в базу данных помещается вопрос и время его создания. Потом из базы данных запрашивается наибольший PolllD (идентификатор опроса) и сохраняется в переменной HighID. Затем все опросы переводятся в неактивное состояние, чтобы сделать единственным активным опросом добавляемый в данный момент опрос (если сделать активными сразу несколько опросов, программа будет работать некорректно, так как ColdFusion и Flash-код, обсуждаемый позже, на это не рассчитаны). В конце мы проходим по всем ответам из соответствующего текстового поля, разделяя их с помощью символа |, и помещаем ответы один за одним в базу данных. Таким образом можно обновить вопросы и ответы нашего опроса. Примите во внимание тот факт, что при обновлении все результаты голосования обнуляются. Следовательно, при добавлении нового опроса постарайтесь сразу указать все возможные ответы, так как вы не сможете потом поменять ответы без удаления текущих результатов опроса.

Опрос: вариант, написанный на ColdFusion

Мы уже подготовили XML-документ, базу данных и конфигурационную программу (admin), осталось написать ColdFusion-код, соединяющий Flash и базу данных, и после этого можно будет заняться непосредственно кодом Flash клиента. В этой главе в основном используется ColdFusion. Выбор этого языка очевиден в силу его простого синтаксиса и возможности связи с базой данных. В нашем случае нет необходимости обращаться к Java или ASP.NET, так как нет проблем с быстродействием, обмен информацией между клиентом и сервером минимален. В следующих разделах мы увидим и разберем четыре сценария ColdFusion, используемые в нашей программе опросов.

Как начать работу?

Хотя некоторые программисты предпочитают сначала написать Flash-клиент (front-end), а потом заниматься серверной частью (back-end) на ColdFusion, Java, ASP.NET, PHP или каком-либо другом языке, но общепринято начинать с написания back-end. Какой именно стиль работы выбрать, зависит от вас, главное, чтобы ваш выбор сделал вашу работу легче, быстрее и эффективнее.

Controller.cfm

Файл Controller.cfm используется для определения нескольких глобальных переменных, динамического построения включений и определения типа транзакции, в нем также содержатся проверки для отлавливания возможных ошибок. Хотя довольно трудно утверждать, что какой-то сценарий более важен, чем другие, отсутствие любого из них приведет к отказу программы. Тем не менее данный сценарий является в каком-то смысле наиболее важным потому, что он контролирует работу всей программы. Чтобы лучше это понять, давайте посмотрим на его содержание.

Откройте файл controller.cfm с вашего CD-ROM-диска.

```
<cfsetting enablecfoutputonly="Yes" showdebugoutput="No" catchexceptionsbypattern="No">
<!-- Установка глобальных значений для переменной имени источника данных и типа базы данных —
>
<cfset DATASOURCE_NAME = "Poll">
<cfset DATABASE_TYPE = "ODBC">
<!-- В случае отсутствия переменной doc создать ее пустой по умолчанию -->
<cfparam name="URL.doc" type="string" default="">
<!-- Если переменная doc пуста, выдать ошибку и прекратить исполнение — >
<cfif URL.doc is "">
    <cfoutput>
        <Response>
            <Status>Error</Status>
            <Data>
                <Message>No XML data sent</Message>
```

```
            </Data>
        </Response>
    </cfoutput>
    <!-- Остановка выполнения -->
    <cfabort>
</cfif>
    <!-- Все заключается в блок try -->
    <cftry>
        <!-- Синтаксический разбор XML-документа -->
        <CF_XMLParser xml = URL.doc output="xmlDoc">
        <!-- Поиск типа транзакции -->
        <cfset transaction = xmlDoc.Request.TransactionType.INNER_TEXT>
            <!-- Динамическое включение (include) файла транзакций -->
        <cfinclude template = "#transaction#Transaction.cfm">
        <!-- Обработчик ошибок базы данных -->
        <cfcatch type="Database">
            <cfoutput>
                <Response>
                <Status>Error</Status>
                <Data>
                    <Message>Error accessing database</Message>
                </Data>
            </Response>
        </cfoutput>
    </cfcatch>
    <!-- Обработчик всех других типов ошибок -->
    <cfcatch type="Any">
        <cfoutput>
            <Response>
                <Status>Error</Status>
                <Data>
                    <Message>Error handling transaction</Message>
                </Data>
        </Response>
    </cfoutput>
    </cfcatch>
    </cftry>
    <!-- Отключение атрибута enablecfoutputonly -->
    <cfsetting enablecfoutputonly="No" showdebugoutput="No" catchexceptionsbypattern="No">
```

Так как код содержит много комментариев, вам, наверно, уже понятно, что он делает, тем не менее давайте проведем его разбор. К сожалению для программистов, ColdFusion часто выводит лишние пробелы. От них можно избавиться с помощью элемента <cfsetting>, установив атрибут ENABLECFOUTPUTONLY равным YES. После этого вводятся две глобальные переменные DATASOURCE_NAME и DATABASE_TYPE. Эти переменные используются во всех сценариях ColdFusion для задания DSN (data source name, название источника данных) и типа базы данных. Далее, если сценарию

не передана doc-переменная, она создается пустой. (Пустое значение doc переменной означает, что запрос на транзакцию отсутствует; в этом случае сценарий прекращает исполнение и возвращает сообщение об ошибке.) Если переменная doc не пуста, значит, был сделан запрос на транзакцию; в этом случае осуществляется синтаксический разбор XML-документа и определяется тип транзакции. (Мы имеем три возможных типа: GetPollData, VoteOnPoll и Has Voted.) Как только определен тип транзакции, соответствующий файл загружается в файл controller.cfm с использованием <cfinclude>. Далее проводится проверка на возможные ошибки базы данных, а потом на любые другие ошибки. В конце отключается атрибут enablecfoutputonly. Как видите, файл controller.cfm очень важен для нашей программы, в нем задается DSN и тип базы данных, определяется тип транзакции, загружается файл, соответствующий типу транзакции, и проводится проверка на возможные ошибки.

GetPollDataTransaction.cfm

Файл GetPollDataTransaction.cfm используется для извлечения всей необходимой информации (вопрос, ответы, общее число голосов и т. д.) из базы данных и ее перевода в XML-формат с целью передачи Flash-клиенту. Давайте посмотрим:

Откройте файл GetPollDataTransaction.cfm с CD-ROM- диска.

```
<!-- Для получения данных опроса выполняется запрос в базу данных -->
<cfquery name="GetPoll" datasource="#DATASOURCE_NAME#" dbtype="#DATABASE_TYPE#">
    Select PollID, PollQuestion, TotalVotes
    From Polls
    Where Active = 1
</cfquery>
<!-- В случае отсутствия активного опроса переменной pollID присваивается значение -1 -->
<cfif GetPoll.RecordCount is 0>
    <cfset pollID = -1>
<cfelse>
    <cfset pollID = GetPoll.pollID>
</cfif>
<!-- Запрос на получение всех ответов активного опроса -->
<cfquery name="GetAnswers" datasource="#DATASOURCE_NAME#" dbtype="#DATABASE_TYPE#">
    Select AnswerID, Answer, Votes
    From Answers
    Where PollID = #pollID#
</cfquery>
<!-- Создание возвращаемого XML-документа -->
<cfoutput>
    <Response>
        <Status>Success</Status>
        <Data>
            <Poll ID="#GetPoll.PollID#" TotalVotes="#GetPoll.TotalVotes#">
                <Question>
                    <Text>#GetPoll.PollQuestion#</Text>
                    <Answers>
```

```
                <cfloop query="GetAnswers">
                    <Answer ID="#AnswerID#" Votes="#Votes#">
                    <Text>#Answer#</Text>
                    </Answer>
                </cfloop>
            </Answers>
        </Question>
    </Poll>
  </Data>
 </Response>
</cfoutput>
```

В самом начале из базы данных с помощью запроса GetPoll извлекается идентификатор активного опроса (PollID), а также соответствующий вопрос и ответы (активный опрос определяется значением active, равным 1; как уже говорилось ранее, это опрос, добавленный в последнюю очередь. Во всех остальных опросах значение active установлено в 0). Затем переменной pollID присваивается значение идентификатора найденного опроса (либо -1 если опрос не найден). Потом из базы данных извлекается идентификатор ответов (AnswerID), сами ответы и общее число поданных голосов. В конце на основе полученных данных создается XML-документ для последующей передачи Flash-клиенту.

VoteOnPollTransaction .cfm

После получения клиентом вопроса, ответов и результатов нужно дать пользователю возможность проголосовать. Файл VoteOnPollTransaction.cfm записывает результат голосования в базу данных и помещает на компьютер пользователя cookie, чтобы запомнить о факте голосования. Давайте перейдем к содержанию файла VoteOnPollTransaction.cfm.

Откройте файл VoteOnPollTransaction.cfm с CD-ROM-диска.

```
<!--- Установим идентификатор ответа (answerID) --->
<cfset answerID = xmlDoc.Request.Data.Vote.AnswerID>
<!--- Выполним запрос на обновление опроса --->
<cfquery name="UpdatePollVotes"
datasource="#DATASOURCE_NAME#" dbtype="#DATABASE_TYPE#">
    update Polls, Answers
        set Polls.TotalVotes = (Polls.TotalVotes + 1 ),
        Answers.Votes = (Answers.Votes + 1 )
    where Polls.Active = Yes and
        Polls.PollID = Answers.PollID and
        Answers.AnswerID = #AnswerID#;
</cfquery>
<!--- Установим cookie на машине пользователя --->
<cfcookie name="hasVoted" value="#pollID#" expires="NEVER">
<cfoutput>
    <Response>
```

```
        <Status>Success</Status>
        <Data>
          <Message>Voted properly</Message>
        </Data>
      </Response>
    </cfoutput>
```

Сначала из XML-документа извлекается идентификатор ответа (AnswerID). Потом общее число голосов, поданных в текущем опросе, увеличивается на 1. Число голосов, поданных за данный ответ, также увеличивается на 1. Затем на компьютер пользователя помещается cookie под названием hasVoted, со значением, равным идентификатору опроса (pollID), и неограниченным временем жизни. В конце создается XML-документ чтобы сообщить пользователю об успешной регистрации его голоса.

HasVotedTransaction.cfm

Можно легко себе представить, что найдется человек, который попытается проголосовать несколько раз и тем самым испортить результаты вашего опроса. Именно поэтому после первого голосования на компьютер пользователя помещается cookie (в файле VoteOnPollTransaction.cfm). Файл HasVotedTransaction.cfm проверяет наличие cookie на компьютере пользователя. Давайте посмотрим на этот файл.

Откройте файл HasVotedTransaction.cfm с CD-ROM-диска.

```
<!-- Выполним запрос на получение идентификатора опроса (pollID) -->
<cfquery name="GetPollID" datasource="#DATASOURCE NAME#" dbtype="#DATABASE TYPE#">
    Select PollID
    From Polls
    Where Active = 1
</cfquery>
<!-- Введем дополнительную переменную со значением идентификатора опроса (pollID) -->
<cfset pollID = GetPollID.PollID>
<!-- Получим значение cookie, со значением по умолчанию -1 -->
<cfparam name="cookie.hasVoted" default="-1">
<!-- Возвратим ответный документ -->
<cfoutput>
    <Response>
        <Status>Success</Status>
        <Data>
        <!-- Если значение cookie не равно идентификатору опроса (pollID), значит, пользователь
        еще не голосовал -->
            <cfif pollID IS cookie.hasVoted>
                <VotedAlready>True</VotedAlready>
            <cfelse>
        <VotedAlready>False</VotedAlready>
    </cfif>
```

```
</Data>
</Response>
</cfoutput>
```

Данный код сначала получает идентификатор активного опроса (PollID). Это значение сохраняется в переменной PollID. Затем запрашивается значение cookie, со значением по умолчанию -1. Потом создается XML-документ. Если значение cookie совпадает с идентификатором активного опроса, то элементу <VotedAlready> присваивается истинное значение, в противном случае — ложное. Обратите внимание, что ColdFusion проверяет не только существование cookie, но и совпадение значения cookie с PollID. Это дает пользователю возможность голосования в последующих опросах, запрещая повторное голосование только по данному вопросу.

Почему не JavaScript?

Так как многие предпочитают JavaScript для работы с cookie, мы поместили на CD-ROM файлы cookie_test.fla, cookie_test.html и cookie.js, показывающие, как установить cookie с помощью JavaScript и Flash. При сравнении соответствующего кода в JavaScript и ColdFusion легко можно увидеть, насколько проще это сделано в ColdFusion.

Flash-клиент (front-end)

Теперь, когда серверная часть закончена, можно перейти к написанию Flash-клиента. Он состоит из одной сцены с несколькими частями.

Почему только одна сцена?

Хотя для такой программы довольно логично иметь несколько сцен, работа только с одной сценой позволяет значительно легче разместить загруженные переменные.
- **Можно загрузить данные несколько раз, но это значительно замедлит работу программы и разочарует ваших пользователей.**

Откройте файл poll.fla с CD-ROM-диска. Наша программа состоит из трех частей. Первая часть загружает данные (рис. 1.3); вторая часть показывает ответы (рис. 1.4); и последняя часть показывает результаты опроса (рис. 1.5). Эти три части вместе и составляют нашу программу опросов.

Рис. 1.3. Загрузка данных опроса

Рис. 1.4. Демонстрация ответов и возможность проголосовать

Рис. 1.5. Демонстрация результатов

Перед тем как начать написание клиента, давайте установим параметры клипа. Размер клипа будет 320x325, цвет фона #FF6600, а параметры публикации (Publish settings) оставим такие, какие заданы по умолчанию в Flash 6. После этого добавим слой с именем actions (действия) и, в первом кадре этого слоя, создадим набор функций для работы с разными типами транзакций. Ниже приведены функции, относящиеся к первому кадру слоя actions.

```
tfinclude "ServerData.as"
#include "CommonTransactionFunctions.as"
// Останавливаем клип
stop();
// Сделаем это один раз
sd = new ServerData();
sd.setMethod(sd.SEND_AND_LOAD);
sd.setLanguage(sd.COLD_FUSION);
sd.setURL("http://www.yourserver.com/Controller.cfm");
// Зададим документ, который будет послан на сервер
sd.setDocOut(buildGetPollDataTransaction());
// Зададим документ, который будет содержать ответ
sd.setDocIn(new XML());
// Зададим функцию (callback), которая будет выполнена при выполнении объекта sd
sd.onLoad = parseGetPollData;
function parseGetPollData() {
    xmlDoc = this.getDocIn();
    worked = wasSuccessful(xmlDoc);
    if (worked) {
        var mainNodes = xmlDoc.firstChild.childNodes;
        var dataNode = mainNodes[1];
        var pollNode = dataNode.firstChild;
        var pollNodeAttributes = pollNode.attributes;
        pollID = pollNodeAttributes["ID"];
        pollTotalVotes = pollNodeAttributes["TotalVotes"];
        var questionNode = pollNode.firstChild;
        var questionNodeChildren = questionNode.childNodes;
        pollQuestion = questionNodeChildren[0].firstChild.nodeValue;
        var pollAnswers = questionNodeChildren[1].childNodes;
        answers = newArray();
        for (i=0; i<pollAnswers.length; i++) {
            var tempAnswer = pollAnswers[i];
            var tempAttributes = tempAnswer.attributes;
            var tempAnswerValue = tempAnswer.firstChild;
            answers["ID_"+i] = tempAttributes["ID"];
            answers["Votes_"+i] = tempAttributes["Votes"];
            answers["Answer_"+i] = tempAnswerValue.firstChild.nodeValue;
        }
        totalAnswers = pollAnswers. length;
    }
    // Перейдем к следующему шагу
    determineVoteStatus();
}
// Выполним это
sd.execute();
function determineVoteStatus() {
```

```
    // Зададим документ, который будет послан на сервер
    sd.setDocOut(buildHasVotedTransaction());
    // Зададим документ, который будет содержать ответ
    sd.setDocIn(new XML());
    // Зададим функцию (callback), которая будет выполнена при выполнении объекта sd
    sd.onLoad = parseHasVoted;
    // Выполним это
    sd.execute();
}
function parseHasVoted() {
    xmlDoc = this.getDocIn();
    trace(xmlDoc);
    if (wasSuccessful(xmlDoc)) {
        var xmlDoc = new XML(pollVotedXML);
        var mainNodes = xmlDoc.firstChild.childNodes;
        var dataNode = mainNodes[1];
        var votedAlreadyNode = dataNode.firstChild;
        votedAlready = votedAlreadyNode.firstChild.nodeValue;
        if (votedAlready) {
            gotoAndStop("results");
        } else {
            gotoAndStop("Vote");
        }
    }
}
// Функция buildGetPollDataTransaction создает
// транзакцию GetPollData
function buildGetPollDataTransaction() {
// Создаем простую GetPollData транзакцию
getPollDataRequest = buildBaseRequest("GetPollData");
// Возвращаем XML-объект
return getPollDataRequest;
}
// Конец объявления функции buildGetPollDataTransaction
// Функция buildHasVotedTransaction создает
// транзакцию HasVotedTransaction
function buildHasVotedTransaction() {
    // Создаем простую HasVoted транзакцию
    hasVotedRequest = buildBaseRequest("HasVoted");
    // Возвращаем XML-объект
    return hasVotedRequest;
}
// Конец объявления функции buildHasVotedTransaction
// Функция buildVoteOnPollTransaction создает
// транзакцию VoteOnPollTransaction
function buildVoteOnPollTransaction(pollID, answerID) {
```

```
    // Создаем простую VoteOnPoll транзакцию
    voteRequest = buildBaseRequest("VoteOnPoll");
    // Ищем узел XML-документа
    dataNode = findDataNode(voteRequest);
    // Создаем новый узел для голоса пользователя
    voteNode = tempDoc.createElement("Vote");
    // Получаем атрибуты этого узла
    voteAttributes = voteNode.attributes;
    // Устанавливаем значения атрибутов
    voteAttributes.PollID = pollID;
    voteAttributes.AnswerID = answerID;
    // Добавляем узел с голосом пользователя в найденный ранее узел
    dataNode.appendChild(voteNode);
    // Возвращаем запрос на транзакцию
    return voteRequest;
}
// конец объявления функции buildVoteOnPollTransaction
```

Не беспокойтесь: этот код не так сложен, как может показаться с первого взгляда. Приведенные выше функции играют ключевую роль в оставшейся части нашего Flash-клиента, так что постарайтесь тщательно прочитать все объяснения. Каждому типу транзакций соответствует своя функция. В самом начале в Flash- систему загружаются файлы ServerData.as и CommonTransactionFunctions.as. Затем клип останавливается; переход к другому кадру будет произведен после того, как мы определим, голосовал ли уже пользователь в текущем опросе. Потом устанавливаются тип метода, язык сервера и URL управляющего файла. Также устанавливается XML-документ, который будет передан на сервер, и ответный документ. Событию onLoad объекта sd (ServerData) присваивается значение parseGetPollData, так что при выполнении объекта sd будет вызвана функция parseGetPollData. Затем создается первая функция, parseGetPollData. В этой функции: если Flash успешно загружает XML-документ, то переменной pollID присваивается идентификатор текущего опроса, переменной pollTotalVotes — общее число голосов, поданных в текущем опросе, а переменной pollQuestion — значение текущего вопроса. Затем создается массив под названием answer, который будет содержать все ответы опроса, их идентификаторы и число голосов, поданных за каждый ответ. Переменной totalAnswers присваивается длина переменной pollAnswers (для последующего использования при определении количества копий клипа). В конце вызывается функция determineVoteStatus. Затем выполняется объект sd (выполнение объекта sd означает проведение какой-либо транзакции, в данном случае GetPollData). Если вы вернетесь к нашему плану написания приложения, то вы увидите, что все идет в соответствии с этим планом.

Следующая функция determineVoteStatus. В ней сначала устанавливается XML-документ для передачи на сервер и XML-документ для получения ответа. Затем событию onLoad объекта sd присваивается значение parseHasVoted с целью последующего выполнения функции parseHasVoted. С этими новыми настройками объект sd снова вызывается на исполнение.

Третья функция, parseHasVoted, сначала проверяет получение документа с сервера. В случае успеха переменной votedAlready присваивается истинное или ложное значение, в зависимости от того, голосовал ли уже пользователь в текущем опросе. Если пользователь уже голосовал, программа переходит к кадру результатов для демонстрации текущих результатов. В противном случае осуществляется переход к кадру голосования. Как вы можете видеть, все идет по плану. В данный момент мы закончили со вторым пунктом из первого и второго списка задач.

Функция buildGetPollDataTransaction просто создает транзакцию getPollData и возвращает XML-документ.

Затем следует функция buildHasVotedTransaction. Она создает транзакцию hasVoted и возвращает XML-документ.

Шестой и последней функцией является buildVoteOnPollTransaction. Ей передается два параметра: pollID и answerID. Сначала эта функция создает транзакцию VoteOnPoll. Затем проводится поиск узла[2] в файле VoteOnPollTransaction. Далее создается новый узел с именем Vote (голос) и двумя атрибутами, pollID и answerID. Этот узел добавляется в найденный ранее узел. В конце функция возвращает запрос на транзакцию.

Мы разобрали довольно много составляющих кода! Если у вас осталось какое-то непонимание того, как работают описанные выше функции, я настоятельно советую вернуться и перечитать комментарии и объяснения. Эти функции являются основой нашей программы; их непонимание приведет к непониманию всего остального кода.

Далее создается второй слой под названием loading (загрузка). В первом и единственном ключевом кадре (keyframe) этого слоя размещается текст "Poll data is loading, please be patient" ("Пожалуйста, подождите пока идет загрузка данных опроса"). Это означает, что перед переходом ,к следующему кадру пользователь получает сообщение о загрузке данных.

После этого добавляется слой под названием voteElements, с ключевым кадром, установленным на второй кадр. В этом новом ключевом кадре создается клип, содержащий текстовое поле под названием answer и невидимую кнопку со следующими действиями (actions):

```
on (release) {
    _root.answer = this.answer;
    _root.answerID = this.answerID;
}
```

Эти действия присваивают переменной answer, принадлежащей к основной монтажной линейке (main timeline), значение переменной answer из клипа, то же самое относится к переменной answerID.

Клип перетаскивается на рабочее поле; созданной таким образом копии клипа (или символу, instance) дается имя Vote. Затем добавляется текст "Your current

2. Здесь и далее имеются ввиду узлы XML документа. - *Примеч. науч. ред.*

answer is:" ("Ваш текущий ответ:") и рядом с текстом создается текстовое поле под названием answer для демонстрации текущего ответа. Далее под текстовым полем создается кнопка со следующими действиями:

```
on (release) {
    sd.onLoad = buildVoteOnPollTransaction (pollID, answerID);
    sd.setDocOut(buildVoteOnPollTransaction (pollID, answerID));
    sd.execute();
    gotoAndStop("results");
}
```

Эти действия присваивают событию onLoad значение buildVoteOnPollTransaction, используемое для обновления результатов опроса. Затем XML-документу (который будет послан на сервер) присваивается значение, возвращаемое функцией buildVoteOnPollTransaction, так как эта функция создает XML-документ с выбранным в данный момент ответом. Далее выполняется объект sd, что приводит к отправке на сервер XML-документа, содержащего pollID и answerID, который будет там обработан кодом из файла VoteOnPollTransaction.cfm. В конце осуществляется переход к кадру результатов для показа текущего подсчета голосов.

Создаем еще одну кнопку со следующими действиями:

```
on (release) {
    gotoAndPlay("results");
}
```

Как вы можете легко убедиться, эта кнопка сразу переводит пользователя к кадру результатов, пропуская этап голосования.

На слое actions добавляется второй ключевой кадр со следующим сценарием действий:

```
loop_length = totalAnswers;
// Создание копий клипа
ystart = vote._y;
for (И); i<loop_length; i++) {
    duplicateMovieClip("vote", "vote"+i, i);
    setProperty("vote"+i, _y, ystart+20*i);
    set("vote"+i+".answer", answers["Answer_"+i]);
    set("vote"+i+".answerID", answers["ID "+i]);
}
stop();
```

В этом сценарии сначала переменной loopjength (длина цикла) присваивается значение переменной totalAnswers (которая была задана в первом кадре и содержит общее число ответов). Затем переменной ystart присваивается у-координата клипа vote. Теперь пора скопировать клип, по одной копии на каждый ответ. Для этого мы используем цикл с переменной цикла i, меняющейся от 0 до loop_length-l. В каждом проходе цикла создается новая копия клипа vote с именем "vote"+i, в случае трех ответов мы получим три клипа с именами: vote0, vote1

и vote2. Каждый новый клип имеет у-координату на 20 пикселов больше, чем предыдущий. Переменная answer устанавливается равной элементу массива answers["Answer_"+i], а переменная answerID устанавливается равной элементу массива answers["ID_"+i]. В конце клип останавливается с помощью функции stop.

В заключение добавляем во втором кадре (рис 1.6) новый слой под названием question (вопрос). В нем создается текстовое поле с переменной под именем pollQuestion. В этом поле будет показан вопрос опроса.

Рис. 1.6. Второй кадр нашей программы.

Также в слое question добавляем третий кадр. Далее добавляем слой с именем resultElements и ключевым кадром в третьем кадре. В этом ключевом кадре создаем текстовое поле с переменной totalvotes; в этом поле будет показано общее число голосов опроса. Затем создается клип для показа результатов опроса. В этом клипе два текстовых поля: answer (для показа возможных ответов) и votes (для показа числа голосов, поданных за конкретный ответ). В клипе также есть рисунок, масштабированный в ширину от 7 до 115 пикселов на протяжении 100 кадров. У каждого слоя в клипе есть ключевой кадр на кадрах 1 и 100, чтобы текстовые поля не исчезали при масштабировании рисунка (рисунок используется для графического отображения числа голосов за каждый ответ в виде горизонтальных полос). Возвращаемся к основной монтажной линейке, создаем копию клипа путем его перетаскивания в рабочее поле и называем эту копию result_mc. Затем добавляем ключевой кадр в третьем кадре слоя actions со следующим сценарием действий:

```
totalVotes = "Votes:"+pollTotalVotes;
loop_length = totalAnswers;
// копируем клип
ystart = result_mc._y;
for (i=0; i<loop_length; i++) {
    duplicateMovieClip("result_mc", "result_mc"+i, i);
    setProperty("result_mc"+i, _y, ystart+20*i);
    set("result_mc"+i+".answer", answers["Answer_"+i]);
    set("result_mc"+i+".votes", "Votes:"+answers["Votes_"+i]);
```

```
        set("result mc"+i+".percent", answers["Votes "+i]/pollTotalVotes*100);
    }
    stop();
```

Вначале переменной totalVotes присваивается строка "Votes:" плюс общее число голосов в опросе. Так же как и в ключевом кадре 2, переменной loop_length присваивается значение totalAnswers (количество ответов). Затем переменной ystart присваивается y-координата клипа result_mc. После этого создаем несколько копий клипа result_mc, по одному на каждый ответ. Значение переменной цикла i меняется от 0 до loop_length–1. В каждом проходе цикла создается копия клипа под именем "result_mc"+i. Y-координата каждой новой копии на 20 пикселов больше, чем у предыдущей. Переменной клипа answer присваивается значение элемента массива answers["Answer_"+i], а переменной votes значение "Votes:" +answers["Votes_"+i]. Переменной percent присваивается процент голосов, поданных за данный ответ, относительно общего числа голосов.

В завершение возвращаемся к клипу result_mc и добавляем к первому ключевому кадру следующий код:

```
stop();
percent = Math.round(percent);
gotoAndStop(percent);
```

Клип останавливается, потом вычисляется округленное значение переменной percent и осуществляется переход к соответствующему кадру. Таким образом рисуется горизонтальная полоса, демонстрирующая результат. На рис. 1.7 показан третий кадр опроса.

Рис. **1.7.** Третий кадр опроса

Мы разобрали все ключевые моменты, связанные с написанием Flash клиента для нашей программы опросов. Не рассмотренными остались только рисунки и графические украшения.

Другие варианты программы опросов

Нашу программу опросов можно очень легко настроить на работу с другим языком сервера, таким, как Java или ASP.NET. Существует множество возможных причин для изменения языка сервера. Например, если вы знаете Java лучше, чем ColdFusion, вам может быть удобней работать с Java-вариантом. Следующие два раздела рассматривают варианты программы опросов, использующие на стороне сервера Java или ASP.NET.

Java

Для подключения к Java-коду, находящемуся на CD-ROM-диске, достаточно изменить две строчки кода. Найдите в первом кадре слоя actions следующие строчки:

```
sd.setLanguage(sd.COLD_FUSION);
sd.setURL("http://www.yourserver.com/Controller.cfm");
```

и замените их на

```
sd.setLanguage(sd.JAVA);
sd.setURL("http://www.yourserver.com/TransactionController");
```

ASP.NET

Так же как и в предыдущем случае, для подключения к ASP.NET коду с CD-ROM диска достаточно поменять следующий код:

```
sd.setLanguage(sd.COLD_FUSION);
sd.setURL("http://www.yourserver.com/Controller.cfm");
```

на:

```
sd.setLanguage(sd.ASPNET);
sd.setURL("http://www.yourserver.com/TransactionController.asp");
```

Возможные изменения в программе опросов

Задумались, что можно сделать еще? Есть несколько возможных улучшений. Например, если ответов будет 20, многие из них станут невидимы. Для решения этой проблемы можно увеличить рабочее поле или добавить элемент прокрутки. Если допускаются ответы из нескольких слов, то можно использовать новое свойство (в среде MX) текстового поля TextField._width для динамического изменения длины текстовых полей с ответами. Возможно, также придется поменять дизайн графических элементов, чтобы избежать наложения на них текстовых полей ответов, иначе программа будет выглядеть неорганизованно и некрасиво. Также можно поменять способ отображения результатов. Вместо горизонтальной гистограммы (bar graph) можно попробовать секторную диаграмму (pie chart), используя либо новую функцию среды MX MovieClip.CurveTO, либо соответствующий компо-

нент с сайта обмена компонентами http://www.macromedia.com/exchange/. Надеюсь, что вышеперечисленные изменения помогут вам улучшить программу опросов в соответствии с вашими требованиями. Даже если программа уже удовлетворяет всем вашим требованиям, вы можете просто поэкспериментировать с целью обучения, так как это лучший способ чему-нибудь научиться.

Заключение

Мы использовали программу опросов в качестве введения в мир написания динамических программ в среде MX на профессиональном уровне. Идеи и методы, описанные в данной главе, используются на более высоком уровне в остальных главах, поэтому их понимание необходимо. В этой главе мы рассмотрели, как объединить ColdFusion, XML и Flash; ASP.NET и Flash; Java и Flash. Вдобавок к этому вы научились планировать свою работу над программой. Я хочу подчеркнуть, что сценарии ColdFusion, ASP.NET, Java и Flash были написаны по одному и тому же плану. Когда у вас есть хороший план, написание кода становится гораздо легче. Это пример того, как немного времени, потраченного с умом в начале проекта, может сохранить много времени в конце. Помните, знание - сила, и пользуйтесь знанием, приобретенным в этой главе, для освоения более сложных программ в следующих главах.

2. Гостевая книга

Автор Макс Ошман (Max Oshman)

Гостевая книга является сегодня, наверно, самой популярной слагающей Интернета и, возможно, одной из самых полезных. Это факт: люди любят общение и самовыражение, и гостевая книга дает для этого хорошую возможность.

В этой главе мы разберем и обсудим гостевую книгу, написанную на ColdFusion, ASP.NET и Java, с использованием Flash в качестве клиента (front-end). Также затрагиваются некоторые из новых возможностей среды Flash MX, например такие, как компоненты. Без лишних разговоров давайте перейдем прямо к делу.

Что это такое?

Гостевая книга дает вашим пользователям превосходную возможность оставить на вашем сайте комментарии, вопросы и предложения. Книга становится местом для встреч, где ваши пользователи могут познакомиться, обсудить последние новости и спорные вопросы. Гостевая книга полезна для любого сайта, независимо от темы. Наш вариант гостевой книги позволяет пользователям не только оставить сообщение, но и сопроводить его картинкой. Это очень интересный проект, который наверняка будет пользоваться большой популярностью у посетителей вашего сайта.

Каким образом это будет работать?

Давайте посмотрим, как будет работать наша программа. У пользователей будет возможность поместить новое сообщение, а также просмотреть уже существующие. Одна из особенностей книги — возможность добавить к сообщению картинку. На сервере эти данные никак не используются, они только хранятся в базе данных в виде XML-документа.

Порядокработы с сервером со стороны клиента

1. Клиент запрашивает у сервера все записи на определенную дату.
2. Flash размещает новую запись, если у пользователя возникло желание добавить запись.

Детали взаимодействия клиента с сервером

1. Получение записей.
   а) Посылается запрос "GetEntries" на транзакцию.
   б) Сервер возвращает данные в формате XML.
2. Размещение новой записи.
   а) Посылается XML документ со всеми необходимыми данными и запрос "PostEntry" на транзакцию.
   б) Сервер сохраняет данные в базе данных и возвращает сообщение об успехе или ошибке.

База данных

Так же как и проект из гл. 1, наш проект использует базу данных. Давайте взглянем на нашу базу данных (рис. 2.1), в ней только одна таблица под названием Entries (записи).

Откройте файл Guestbook-97.mdb с CD-ROM-диска.

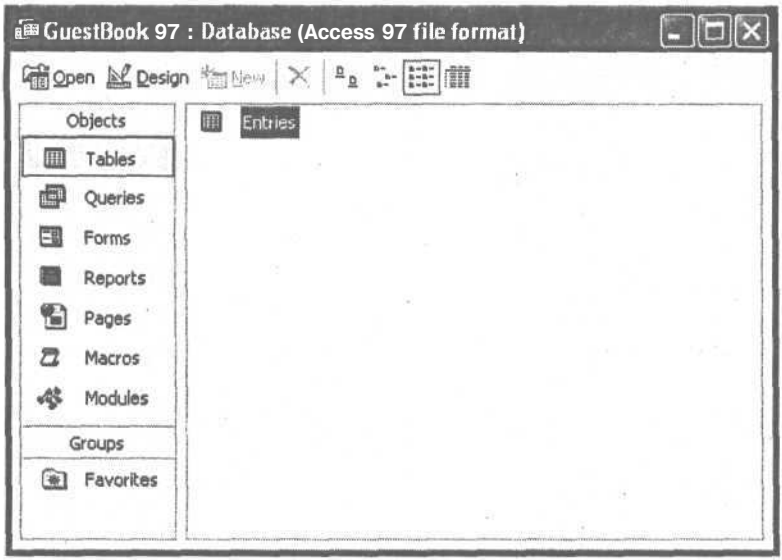

Рис. 2.1. База данных, необходимая для работы опроса, состоит из одной таблицы.

Записи

Это единственная таблица в базе данных. В ней хранится вся информация, внесенная пользователем: имя, электронный адрес (email), заголовок и сообщение. Вдобавок к этому в таблице есть колонка под названием Drawing (рисунок), содержащая информацию о том, какой рисунок был выбран для сопровождения

сообщения. В базе данных также хранится дата размещения сообщения и IP-адрес компьютера пользователя.

XML-документ

Так же как ив случае программы опросов, обмен данными между Flash-клиентом и сценариями на сервере проходит в формате XML. Фрагменты из данного XML-документа появляются в разных местах ColdFusion-Java- и ASP.NET-кода.

Откройте файл Guestbook.xml с CD-ROM диска.

Ниже приведен XML-документ, используемый во всех трех вариантах гостевой книги (написанных на ColdFusion, ASP.NET и Java).

```
<Blah>
<!- Запрос на получение записей из гостевой книги –>
<Request>
   <TransactionType>GetEntries</TransactionType>
   <Data />
</Request>
<!- Ответ на предыдущий запрос ->
<Response>
   <Status>Success</Status>
   <Data>
      <Entries>
         <Entry ID="" ShowEmail="True|False">
               <Name></Name>
            <EmailX/Email>
            <Subject></Subject>
            <Message></Message>
            <imagedata></imagedata>
         </Entry>
      </Entries>
   </Data>
</Response>
<!– Запрос на размещение записи –>
<Request>
   <TransactionType>PostEntry</TransactionType>
   <Data>
      <Name></Name>
      <Email></Email>
      <Subject></Subject>
      <Body></Body>
      <ImageData></ImageData>
   </Data>
</Request>
<!- Ответ на предыдущий запрос ->
```

```
<Response>
   <Status>Success</Status>
   <Data>
      <Message>Post entered</Message>
   </Data>
</Response>
<!- Ответ в случае возникновения ошибки ->
<Response>
   <Status>Error</Status>
   <Data>
   <Message>broke</Message>
   </Data>
</Response>
</Blah>
```

Гостевая книга: вариант, написанный на ColdFusion

Мы подготовили XML-документ и базу данных, теперь можно заняться написанием сценариев сервера. Основным языком этой главы является ColdFusion, так как в нем можно легко подключиться к базе данных. Так как от нашей программы не требуется работы в реальном времени (real-time) (в отличие от программ мгновенного обмена сообщениями (instant messenger) или интерактивного общения (chat room)), быстродействие для нас не проблема. В следующих разделах мы увидим и разберем три сценария ColdFusion, используемые в нашей гостевой книге.

Controller.cfm

Файл Controller.cfm такой же, как и в предыдущей главе. Он используется для определения нескольких глобальных переменных, динамического построения включений и определения типа транзакции, в нем также содержатся проверки для отлавливания возможных ошибок. Давайте освежим в памяти его содержание.

Откройте файл controller.cfm с вашего CD-ROM-диска.

```
<cfsetting enablecfoutputonly="Yes" showdebugoutput="No" catchexceptionsbypattern="No">
<!-- Установка глобальных значений для переменной имени источника данных и типа базы данных -->
<cfset DATASOURCE_NAME = "Poll">
<cfset DATABASE_TYPE = "ODBC">
<!-- В случае отсутствия переменной doc создать ее пустой по умолчанию -->
<cfparam name="URL.doc" type="string" default="">
<!-- Если переменная doc пуста, выдать ошибку и прекратить исполнение -->
<cfif URL.doc is "">
   <cfoutput>
      <Response>
```

```
            <Status>Error</Status>
            <Data>
               <Message>No XML data sent</Message>
            </Data>
         </Response>
      </cfoutput>
      <!-- Остановка выполнения -->
      <cfabort>
   </cfif>
   <!-- Все заключается в блок try -->
   <cftry>
      <!-- Синтаксический разбор XML документа -->
      <CF_XMLParser xml = URL.doc output="xmlDoc">
      <!-- Поиск типа транзакции -->
      <cfset transaction = xmlDoc.Request.TransactionType.INNER_TEXT>
      <!-- Динамическое включение (include) файла транзакций -->
      <cfinclude template = "#transaction#Transaction.cfm">
      <!-- Обработчик ошибок базы данных -->
      <cfcatch type="Database">
         <cfoutput>
            <Response>
               <Status>Error</Status>
               <Data>
                  <Message>Error accessing database</Message>
               </Data>
            </Response>
         </cfoutput>
      </cfcatch>
   <!-- Обработчик всех других типов ошибок -->
   <cfcatch type="Any">
      <cfoutput>
         <Response>
            <Status>Error</Status>
            <Data>
               <Message>Error handling transaction</Message>
            </Data>
         </Response>
         </cfoutput>
      </cfcatch>
      </cftry>
   <!-- Отключение атрибута enablecfoutputonly -->
   <cfsetting enablecfoutputonly="No" showdebugoutput="No" catchexceptionsbypattern="No">
```

Так как этот код уже объяснялся ранее, просто перечислим вкратце основные моменты. Сначала отключаем вывод лишних пробелов, затем задаем две переменные, одну для имени источника данных (DSN), а другую для типа базы данных. Проверяем, что серверу был послан XML-документ, содержащий запрос на

транзакцию. Если это не так, выводится сообщение об ошибке, в противном случае определяется тип транзакции и загружается соответствующий сценарий ColdFusion. В конце проводится проверка на возможные ошибки.

GetEntriesTransaction.cfm

Файл GetEntriesTransaction.cfm используется для извлечения большей части данных гостевой книги (все за исключением колонок drawing (рисунок), createdate (дата создания) и ipaddress (IP-адрес)) из базы данных и их перевода в XML формат с целью передачи Flash-клиенту. Давайте взглянем.

Откройте файл GetEntriesTransaction.cfm с CD-ROM-диска.

```
<!-- Для получения данных гостевой книги выполняется запрос в базу данных -->
<cfquery name="GetGuestbook" datasource="#DATASOURCE_NAME#"
dbtype="#DATABASE_TYPE#">
    Select EntryID, name, email, showemail, subject, message, drawing
    From Entries
</cfquery>
<!-- Если записи отсутствуют, переменной entryID присваивается -1 -->
<cfif GetGuestbook.RecordCount is 0>
    <cfset entryID = -1>
<cfelse>
    <cfset entryID = GetGuestbook.entryID>
</cfif>
<!-- Создание XML-документа, который будет возвращен -->
<cfoutput>
    <Response>
        <Status>Success</Status>
        <Data>
            <cfloop query="GetGuestbook">
                <Entries>
                <Entry ID="#EntryID#" ShowEmail="#ShowEmail#">
                        <Name>#name#</Name>
                        <Email>#email#</Email>
                        <Subject>#subject#</Subject>
                        <Message>#message#</Message>
                        <imagedata>#drawing#</imagedata>
                    </Entry>
                </Entries>
            </cfloop>
        </Data>
    </Response>
</cfoutput>
```

Этот сценарий трудно упростить еще больше. Сначала из базы данных извлекаются необходимые нам данные. Затем переменной EntryID присваивается

число полученных записей (-1 в случае отсутствия записей). В конце создается XML-документ для передачи данных Flash-клиенту. Этот документ содержит значения EntryID, ShowEmail, а также имя, электронный адрес, тему и текст записи.

PostEntryTransaction.cfm

Этот сценарий посылает все данные, полученные от пользователя, на запись в базу данных. Давайте рассмотрим его более подробно.

Откройте файл PostEntryTransaction.cfm с CD-ROM-диска.

```
<!-- Промежуточная переменная для хранения ссылки на узел XML-документа -->
<cfset dataNode = xmlDoc.Request.Data>
<!-- Получаем имя -->
<cfset name = dataNode.Name.INNER_TEXT>
<!-- Получаем электронный адрес -->
<cfset email = dataNode.Email.INNER_TEXT>
<!-- Получаем значение showemail (определяет, будетли электронный адрес скрытым или нет) —
>
<cfset showemail = dataNode.ShowEmail.INNER_TEXT>
<!-- Получаем тему -->
<cfset subject = dataNode.Subject.INNER_TEXT>
<!-- Получаем текст записи -->
<cfset message = dataNode.Message.INNER_TEXT>
<!-- Получаем данные о рисунке -->
<cfset imagedata = dataNode.imagedata.INNER_TEXT>
<!-- Выполняем запрос на сохранение данных в базе данных-->
<cfquery name="PostMessage" datasource="#DATASOURCE_NAME#" dbtype="#DATABASE_TYPE#">
    Insert into Entries
        (name, email, showemail, subject, message, drawing, createdate, IPAddress)
        values
            ('#name#', '#email#', '#showemail#', '#subject#', '#message#', '#imagedata#', #Now()#,
            '#REMOTE_ADDR#')
        </cfquery>
<!- Ответ на PostEntry транзакцию->
<Response>
    <Status>Success</Status>
    <Data>
        <Message>Message Recorded</Message>
    </Data>
</Response>
<!- Ответ в случае возникновения ошибки ->
<Response>
    <Status>Error</Status>
    <Data>
        <Message>There was an error, please try again</Message>
    </Data>
```

Сначала из XML-документа извлекаются все переданные значения и присваиваются соответствующим переменным. Затем мы сохраняем их в базе данных. В конце создается и возвращается XML-документ с сообщением об успехе или ошибке выполнения сценария.

Flash-клиент (front-end)

Наконец-то мы достигли того момента, когда можно заняться программированием в среде Flash! База данных и сценарии ColdFusion, сохраняющие и извлекающие информацию из базы данных, являются хорошим фундаментом, и теперь мы можем собрать все части вместе с помощью Flash-клиента в качестве front-end.

Откройте файл Guestbook.fla с CD-ROM-диска. Наша программа состоит из трех частей. Первая часть загружает записи (рис. 2.2); вторая часть показывает записи (рис. 2.3); и последняя часть позволяет разместить новую запись (рис 2.4).

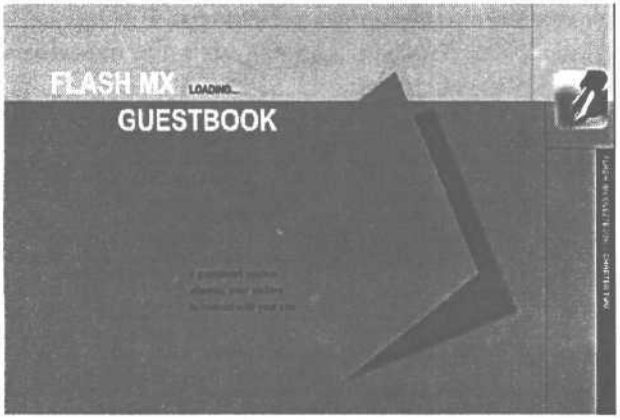

Рис. 2.2. Загрузка записей

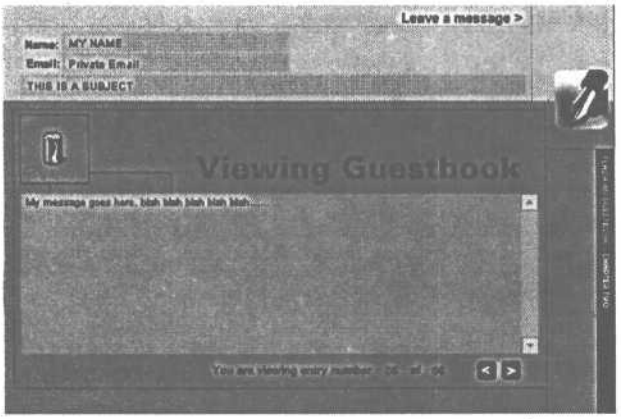

Рис. 2.3. Демонстрация записей

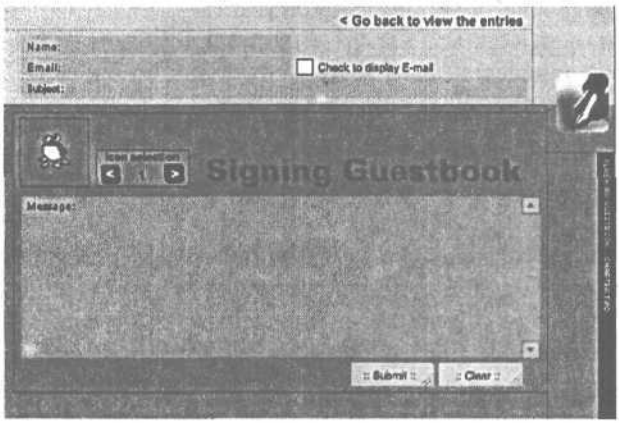

Рис. 2.4. Кадр, позволяющий добавить новую запись

Сначала настроим параметры клипа. Размер оставим таким, какой стоит по умолчанию, 550x400; установим серый фон (#666666) и скорость 120 кадров в секунду (FPS, частота смены кадров). После этого добавляем слой под названием actions (действия) со следующими действиями:

```
#include "ServerData.as"
#include "CommonTransactionFunctions.as"
// Останавливаем клип
stop();
// Сделаем это один раз
sd = new ServerData();
sd.setMethod(sd.SEND_AND_LOAD);
sd.setLanguage(sd.COLD_FUSION);
sd.setURL("http://www.yoururl/controller.cfm");
sd.setDocOut(buildGetEntriesTransaction());
// Зададим документ, который будет содержать ответ
sd.setDocIn(new XML());
// Зададим функцию (callback), которая будет выполнена при выполнении объекта sd
sd.onLoad = parseGetEntries;
// Получим записи из гостевой книги и упорядочим их
function parseGetEntries() {
    xmlDoc = this.getDocIn();
    worked = wasSuccessful(xmlDoc);
    if (worked) {
        var mainNodes = xmlDoc.firstChild.childNodes;
        var dataNode = mainNodes[1];
        var EntriesNode = dataNode.childNodes;
        var EntryNodeChildren = EntryNode.childNodes;
        // Создадим массив для хранения всех данных из гостевой книги
        entriesdata = new Array();
        for (i=0; i<EntriesNode.length; i++) {
            var tempEntry = EntriesNode[i];
```

```
            var tempAttributes = tempEntry.firstChild.attributes;
            var tempEntryValue = tempEntry.firstChild;
            entriesdata["ID_"+i] = tempAttributes["ID"];
            entriesdata["ShowEmail_"+i] = tempAttributes["ShowEmail"];
            entriesdata["Name_"+i] = tempEntryValue.firstChild.firstChild;
            entriesdata["Email_"+i] = tempEntryValue.firstChild.nextSibling.firstChild;
            entriesdata["Subject_"+i] = tempEntryValue.firstChild.nextSibling.nextSibling.firstChild;
            entriesdata["Message_"+i] = tempEntryValue.firstChild.nextSibling.nextSibling.nextSib-
            ling.firstChild;
            entriesdata["imagedata_"+i] = tempEntryValue.firstChild.nextSibling.nextSibling.nextSib-
            ling.nextSibling.firstChild;
        }
            //покажем результаты
            total_entries = EntriesNode.length-1;
            i = i-1;
            _root.name = entriesdata["Name_"+i];
            _root.showemail = entriesdata["ShowEmail_"+i];
            _root.subject = entriesdata["Subject_"+i];
            _root.message = entriesdata["Message_"+i];
            root.images.gotoAndStop(parseFloat(entriesdata["imagedata_"+i]));
            // Если пользователь пожелал скрыть свой электронный адрес, заменим его строкой "Pri-
            vate Email"
            if (_root.showemail == "O") {
                _root.email = "Private Email";
            } else {
                _root.email = entriesdata["Email "+i];
            }
            gotoAndStop(2);
        }
}
sd.execute();
function buildGetEntriesTransaction() {
    GetEntriesRequest = buildBaseRequest("GetEntries");
    return GetEntriesRequest;
}
function postEntry() {
    // Зададим документ, который будет послан на сервер
    sd.setDocOut(buildPostEntryTransaction());
    // Зададим документ, который будет содержать ответ
    sd.setDocIn(new XML());
    // Зададим функцию для выполнения после загрузки данных
    sd.onLoad = parseGetEntries;
    // Выполним это
    sd.execute();
}
function buildPostEntryTransaction() {
```

```
    postRequest = buildBaseRequest("PostEntry");
    // Получаем узел XML-документа
    dataNode = findDataNode(postRequest);
    //Создаем новые XML-элементы
    nameNode = postRequest.createElement("Name");
    showemailNode = postRequest.createElement("ShowEmail");
    emailNode = postRequest.createElement("Email");
    subjectNode = postRequest.createElement("Subject");
    messageNode = postRequest.createElement("Message");
    imagedataNode = postRequest.createElement("imagedata");
    // Создаем текстовые XML-узлы
    var nameValue = postRequest.createTextNode(name);
    var showemailValue = postRequest.createTextNode(showemail);
    var emailValue = postRequest.createTextNode(email);
    var subjectValue = postRequest.createTextNode(subject);
    var messageValue = postRequest.createTextNode(message);
    var imagedataValue = postRequest.createTextNode(imagedata);
    // Добавляем XML-узлы друг к другу
    nameNode.appendChild(nameValue);
    showemailnode.appendChild(showEmailValue);
    emailnode.appendChild(emailValue);
    subjectnode.appendChild(subjectValue);
    messageNode.appendChild(messageValue);
    imagedataNode.appendChild(imagedataValue);
    // Добавляем XML-узлы к полученному ранее узлу XML-документа
    dataNode.appendChild(nameNode);
    dataNode.appendChild(emailNode);
    dataNode.appendChild(showEmailNode);
    dataNode.appendChild(subjectNode);
    dataNode.appendChild(messageNode);
    dataNode.appendChild(imagedataNode);
    // Возвращаем запрос на транзакцию
    gotoAndPlay(1);
    return postRequest;
}
```

Приведенные выше функции контролируют работу практически всей нашей программы. Перед тем как создавать функции, нужно позаботиться о некоторых других вещах. В начале загружаются файлы ServerData.as и CommonTransaction-Functions.as. Затем устанавливаем несколько характеристик нашей программы, таких, как способ работы с данными (SEND_AND_LOAD, отправка и загрузка), используемый язык (ColdFusion) и местонахождение (URL) управляющего файла controller.cfm.

Теперь можно осуществить первоначальную загрузку данных. Для этого вызываем функцию buildGetEntriesTransaction для отправки на сервер запроса GetEntries на транзакцию. Затем полученные данные реорганизуются с помощью функции

parseGetEntries. Далее проверяем, что данные получены успешно, и выделяем несколько основных узлов XML-документа для последующей реорганизации. Создаем массив под именем entriesdata для хранения всей полученной информации из гостевой книги (ID, ShowEmail, Name, Subject, Message и imagedata). Присваиваем соответствующие значения переменным Name, Email, ShowEmail, Subject, Message и imagedata. Мы берем только начальные значения данных, так как гостевая книга показывает по одной записи за раз. Значения присваиваются в данном кадре, чтобы избежать задержки в следующем кадре. Затем проверяем значение переменной ShowEmail, чтобы выяснить, можно ли показывать электронный адрес пользователя открытым текстом (если эта переменная равна 0, вместо электронного адреса выводится текст "Private Message"). В конце переходим ко второму кадру для просмотра гостевой книги.

Вторая функция, buildGetEntriesTransaction, собирает запрос GetEntries, который используется выше для получения с сервера записей из гостевой книги.

Третья функция, postEntry, отправляет на сервер запрос, собранный функцией buildPostEntryTransaction, с целью добавления новой записи. Здесь также создается новый XML-документ, который будет использован сервером для возвращения обработанной информации (после получения сервером новой записи). После загрузки данных снова вызывается функция parseGetEntries для реорганизации полученных данных.

Последняя функция, buildPostEntryTransaction, собирает XML-документ для отправки на сервер. Сначала создается транзакция типа PostEntry. Затем проводится поиск узла данных dataNode, чтобы добавить к нему новые элементы с данными (для последующей отправки серверу (сценарию ColdFusion) с целью сохранения в базе данных). Создаем новые XML-элементы nameNode, showemailNode, emailNode, subjectNode, messageNode и imagedataNode. Далее создаем шесть новых текстовых XML-узлов (TextNodes) nameValue, showemailValue, emailValue, subjectValue, message Value, imagedata Value и добавляем их к элементам. Затем добавляем все элементы к узлу dataNode. В конце переходим к первому ключевому кадру для реорганизации данных (мы проводили реорганизацию и в других местах, но в этом месте она также необходима. Если удалить один из вызовов (call backs) либо переход к первому ключевому кадру, либо вызов функции parseGetEntries), — в некоторых случаях реорганизация данных может не произойти.

Добавляем новый слой под именем loading (загрузка) и в первом ключевом кадре помещаем сообщение пользователю о загрузке данных.

Далее создаем новый слой под названием showmessage. Помещаем во второй кадр ключевой кадр и начинаем размещать на рабочем поле объекты, необходимые для просмотра записей. Добавляем четыре текстовых поля name (имя), email (адрес), subject (тема) и message (сообщение), открываем меню компонентов и создаем копию компонента прокрутки (ScrollBar) путем перетаскивания на рабочее поле. Помещаем этот компонент прокрутки на текстовое поле сообщения, чтобы у пользователя была возможность добавить большое сообщение. Затем добавляем кнопки Next (вперед) и Previous (назад) с целью перемещения между записями. Добавляем к кнопке Next следующие действия:

```
on (release) {
    // Если переменная i не равна общему числу записей,
    // добавляем 1 и обновляем показываемые данные,
    // используя следующий элемент массива entriesdata
    if (_root.total_entries != i) {
        i = i+1;
        _root.name = entriesdata["Name_"+i];
        joot.showemail = entriesdata["ShowEmail_"+i];
        joot.subject = entriesdata["Subject_"+i];
        _root.message = entriesdata["Message_"+i];
        _root.images.gotoAndStop(parseFloat(entriesdata["imagedata_"+i]));
        // Проверяем, должен ли электронный адрес быть скрытым
        if(_root.showemail=="O"){
            _root.email = "Private Email";
        } else {
            _root.email = entriesdata["Email "+i];
        }
    } else {
        _root.i = _root.total_entries;
    }
}
```

Данная кнопка используется для перемещения вперед по записям. Если переменная i не равна общему количеству записей, переходим к следующей записи. Переменным name, showemail, subject, message и imagedata присваиваются соответствующие значения массива entriesdata, в результате чего будет показана следующая по порядку запись. Мы не должны также забыть о возможной анонимности пользователя. Если пользователь пожелал остаться анонимным, вместо его электронного адреса выводится сообщение "Private Email" (анонимное сообщение).

Затем добавляем к кнопке Previous следующие действия:

```
on (release) {
    // Если i не равно 0, уменьшаем значение i
    // и обновляем показываемые данные
    if (_root.i != 0) {
        i = i-1;
        joot.name = entriesdata["Name_"+i];
        joot.showemail = entriesdata["ShowEmail_"+i];
        joot.subject = entriesdata["Subject_"+i];
        _root.message = entriesdata["Message_"+i];
        _root.images.gotoAndStop(parseFloat(entriesdata["imagedata_"+i]));
        // Проверяем анонимность адреса
        if (_root.showemail == "0") {
        _root.email = "Private Email";
    } else {
        joot.email = entriesdata["Email_"+i]; } } }
```

Эти действия аналогичны действиям кнопки Next, только в обратном порядке. Если значение переменной i не равно 0, переходим к предыдущей записи, отображая соответствующие элементы массива entriesdata. В конце проверяем анонимность адреса.

Затем создаем кнопку под заголовком "Leave a message" ("оставить сообщение"), позволяющую перейти к третьему ключевому кадру, где пользователю предоставляется возможность сделать новую запись. Данной кнопке соответствуют следующие действия:

```
on (release) {
name = "Name:"
email = "Email:"
showemail = "0"
subject = "Subject:"
message = "Message:"
gotoAndStop(3);
}
```

Присваиваем каждой переменной значение по умолчанию, а затем переходим к третьему ключевому кадру.

Создаем клип под названием images (картинки). Размещаем в этом клипе пять ключевых кадров для изображений. Здесь будет показываться картинка, выбранная пользователем для сопровождения записи. Копию клипа (instance) также называем images.

Затем создаем еще два текстовых поля, с ассоциированными переменными i и total entries, и размещаем их рядом друг с другом с целью показа общего числа записей и номера текущей записи. Например, если пользователь просматривает девятую из 29 записей, в данных текстовых полях будет написано: "You are viewing entry number 9 of 29" ("Вы видите запись номер 9 из 29").

В конце добавим к слою actions во втором кадре второй ключевой кадр со следующим действием:

```
stop ();
```

Это действие останавливает клип для просмотра записей пользователем. На рис. 2.5 показан окончательный вид второго кадра.

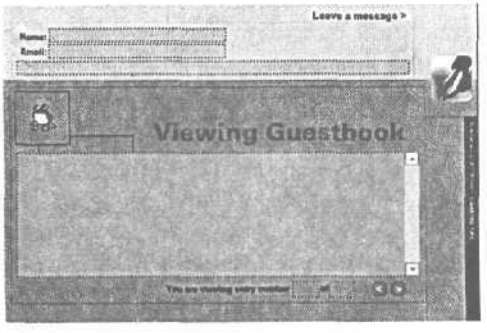

Рис. 2.5. Второй кадр гостевой книги

Добавляем третий слой под названием leavemessage (оставить сообщение) и размещаем в третьем кадре ключевой кадр. В этом ключевом кадре помещаем такие же текстовые поля, как во втором ключевом кадре слоя showmessages, только другого типа, Input Text вместо Dynamic Text, чтобы пользователь мог ввести свои данные.

Затем добавляем возможность анонимности пользователя. Для этого создаем клип с именем check_box и двумя кадрами (в одном стоит галочка, а другой пуст). В первом кадре, без галочки, находится кнопка со следующими действиями:

```
on (release) {
_root.ShowEmail = 1 ;
gotoAndStop(2);
}
```

Переменной ShowEmail присваивается 1 (электронный адрес можно показывать открытым текстом), и осуществляется переход ко второму кадру.

Во втором кадре (с галочкой) находится кнопка со следующими действиями:

```
on (release) {
    _root.ShowEmail = 0;
    gotoAndStop(1);
}
```

Переменной ShowEmail присваивается 0 (анонимный пользователь), затем клип переходит к первому кадру.

Переходим к основной монтажной линейке и создаем копию клипа images (путем перетаскивания на рабочее поле) под названием images. Добавляем кнопки Next (вперед), Previous (назад) и размещаем их под клипом images. С их помощью можно будет переходить между различными картинками при выборе картинки для сопровождения записи. Кнопка Next получает следующие действия:

```
on (release) {
    if (images._currentframe != 5) {
        images.gotoAndStop(images._currentframe + 1);
        _root.imagedata = root.imagedata + 1 ;
    }
}
```

Если текущий кадр не равен 5, переходим к следующему кадру и увеличиваем значение переменной imagedata на 1 (значения этой переменной и текущего кадра совпадают, так как при загрузке этой переменной в среду Flash ее значение используется для установки текущего кадра клипа images).

Кнопка Previous получает следующие действия:

```
on (release) {
    if (images._currentframe != 1 ) {
        images.gotoAndStop(images._currentframe - 1);
        _root.imagedata = root.imagedata - 1 ;
    }
}
```

Эти действия противоположны действиям кнопки Next. Если текущий кадр не является первым, переходим к предыдущему кадру и уменьшаем значение переменной imagedata на 1.

Затем создаем кнопку Submit (разместить) со следующими действиями:

```
on (release) {
postEntry();
}
```

Здесь вызывается функция postEntry, отправляющая внесенные пользователем данные на сервер.

Далее создаем кнопку Clear (очистить) с действиями:

```
on (release) {
name="Name:"
email="Email:"
subject="Subject:"
message="Message:"
}
```

Соответствующим переменным присваиваются значения по умолчанию. На этом мы закончили с третьим кадром. На рис. 2.6 показан конечный результат.

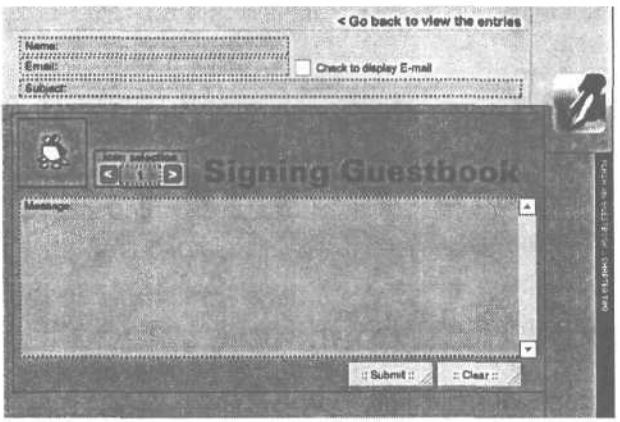

Рис. 2.6. Третий кадр гостевой книги

Как видите, все было довольно просто. Если у вас возникли какие-то сложности, я рекомендую вам снова просмотреть две первых главы и перечитать все места, объясняющие код. Следующие главы будут более трудными, и вам нужно убедиться, что вы полностью разобрались в описанном ранее коде и методах программирования.

Другие варианты гостевой книги

Очень легко настроить гостевую книгу на работу с другим языком сервера, таким, как Java или ASP.NET. Существует множество возможных причин для изменения языка сервера, как уже было упомянуто в гл. 1. Следующие два раздела рассматривают варианты гостевой книги, использующие на стороне сервера Java или ASP.NET.

Java

Для подключения к Java-коду, находящемуся на CD-ROM, достаточно изменить две строчки кода. Найдите в первом кадре следующие строчки:

```
sd.setLanguage(sd.COLD_FUSION);
sd.setURL("http://www.yourserver.com/Controller.cfm");
```

и замените их на:

```
sd.setLanguage(sd.JAVA);
sd.setURL("http://www.yourserver.com/TransactionController");
```

ASP.NET

Так же как и в предыдущем случае, для того чтобы подключиться к ASP.NET-коду с CD-ROM, достаточно поменять код:

```
sd.setLanguage(sd.COLD_FUSION);
sd.setURL("http://www.yourserver.com/Controller.cfm");
```

на

```
sd.setLanguage(sd.ASPNET);
sd.setURL("http://www.yourserver.com/TransactionController.asp");
```

Возможные изменения в гостевой книге

Существует не так много возможных изменений гостевой книги, тем не менее рассмотрим некоторые из них. Для начала, так как гостевая книга может содержать сотни записей и требовать много времени на загрузку, можно вместо простого сообщения о загрузке данных добавить загрузчик (preloader) для иллюстрации процесса загрузки. Для этого можно использовать функции getBytesTotal() и getBytesLoaded(). Если разделить BytesLoaded (объем загруженных данных, в байтах) на BytesTotal (общий объем данных, в байтах) и умножить на 100, мы получим объем загруженных данных в процентах. Также можно поменять набор данных, предлагаемых пользователю для заполнения. Только не забудьте о том, что для работы с новыми данными нужно будет также изменить две основные функции (parseGetEntries и buildPostEntryTransaction). Наиболее сложное изменение, которое приходит на ум, — это добавление маленькой программы, дающей возможность пользователю поместить в базу данных на сервере свое собственное изображение.

Это изображение потом может использоваться Flash-клиентом для сопровождения записей пользователя. Для динамической загрузки изображения в клип images можно использовать следующий код:

```
images.loadMovie("http://www.yourdomain.com/someimage.jpg")
```

Вы также можете придумать и внести любые другие изменения по вашему выбору. Используйте ваше воображение, пределы программирования во Flash еще не достигнуты.

Заключение

Идеи, рассмотренные в данной главе, очень важны. Возможность скрытия какой-нибудь информации о пользователе, например, очень широко применяется в программах типа обмена мгновенными сообщениями (instant messenger). Мы также увидели, как на основе информации от одного пользователя (показывать или нет электронный адрес, выбор картинки) меняется внешний вид программы для другого пользователя. Идеи и методы из этой главы применимы не только к нашей гостевой книге, но и ко многим другим программам. Если у вас возникли трудности с пониманием этой главы, вернитесь и перечитайте непонятные вам места. Когда вы будете готовы, давайте перейдем к следующему проекту — форум (message board).

3. Дискуссионный форум

Автор Эрик Е. Долески (Eric E. Dolecki)

В ваших путешествиях по Интернету вы наверняка уже встречали какой-нибудь дискуссионный форум (message board). Такие форумы обычно являются сложными системами по работе с данными. Они устанавливаются на многих сайтах, чтобы позволить посетителям взаимодействовать между собой, создавая таким образом сообщество пользователей.

Такой форум может заменить во многих аспектах традиционные способы поддержки пользователей, например такие, как справочный стол (help desk[1]). Он позволяет осуществлять обмен сообщениями на разные темы и в разных потоках.

У меня много раз возникали проблемы во время написания Flash-программ. Если мне срочно нужна была помощь, я обращался в дискуссионные форумы, посвященные среде Flash. В таких местах много посетителей, многие профессиональные программисты просматривают сообщения и могут ответить на вопросы. После помещения вопроса обычно в течение одного-двух дней кто-нибудь отвечает и корректирует мой код. Это большой успех для такого сайта. До появления среды Flash MX форумы почти всегда состояли из HTML-кода. Теперь их можно размещать повсеместно с помощью Flash MX и кода поддержки на сервере (backend). В данной главе вы увидите, как это делается.

Свойства нашего Rash MX-дискуссионного форума

Форум, описанный в этой главе, обладает многими свойствами, присущими традиционным системам:

- темами дискуссий (определяются администратором, пользователь не может добавить новую тему);
- прочесть сообщения могут все посетители, для добавления нового сообщения или для ответа на старое требуется зарегистрироваться и войти в систему;
- потоками[2] (threads), основанными на темах дискуссий;
- ответами на потоки (сообщений).

1. Системы поддержки пользователей в сети. — *Примеч. науч. ред.*

2. Возможно, некоторым читателям, будет более привычным понятие "ветвей сообщений". — *Примеч. науч. ред.*

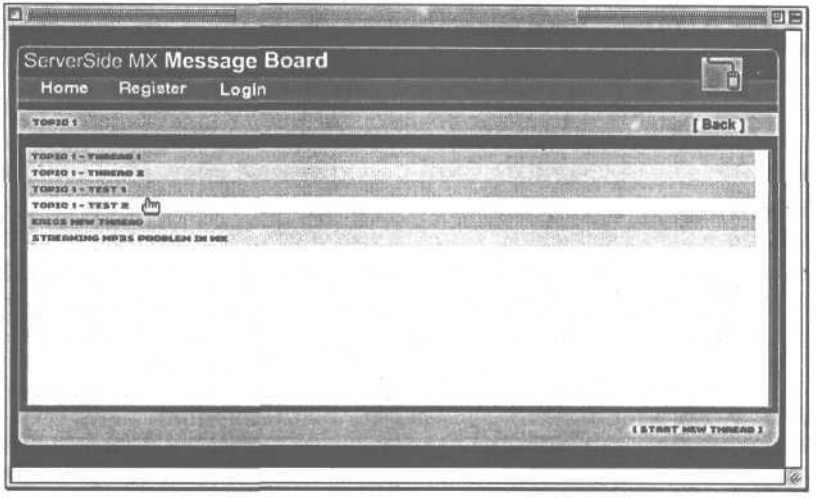

Рис. 3.1. Наш Flash MX форум в действии

Новый компонент Flash MX вместе с новыми возможностями форматирования текста и динамического создания клипов и текстовых полей позволяет нашему форуму оставить далеко позади любую ранее написанную аналогичную Flash-систему. Для администрирования системы можно редактировать записи руками (не рекомендуется) либо написать свою собственную программу управления (admin panel).

Почему именно Flash?

Это хороший вопрос. Подавляющая часть таких систем написана на HTML. Они обычно состоят из множества отдельных ASP-, JSP- или PHP-страниц, выполняющих посредническую роль между пользователем и соответствующими базами данных. Во многих обстоятельствах этого хватает. Однако написание такой системы с нуля может потребовать создания десятков индивидуальных страничек для совершения отдельных операций.

Возможности графического интерфейса (GUI) среды Flash MX позволяют создавать более интегрированные приложения. Отпадает необходимость перехода между многочисленными HTML-страницами, и у пользователя остаются более приятные впечатления после посещения вашего форума.

При помощи надлежащих методов планирования и разработки обычное промежуточное программное обеспечение (middleware), написанное на PHP, ASP и похожих языках, можно заменить единым приложением на сервере. Такая система может даже работать автономно (offline). Однако только эта возможность сама по себе может оказаться недостаточно убедительным аргументом в пользу прм.енения Flash.

Так как Flash используется все более широко, вы будете встречать все больше сайтов и приложений, распространяемых в виде Flash-компонента (в формате SWF).

Представьте, как удобно было бы иметь возможность добавить ваш дискуссионный форум в качестве компонента в другое приложение. Например, во внутренней сети можно установить основанную на Flash программу помощи (Help Desk), содержащую дискуссионный форум, электронную почту, возможность интерактивного общения (chat system) с представителем фирмы и, может быть, даже систему видеоконференций, позволяющую прямое общение с человеком из службы поддержки. И все это реализовано в виде одной большой Flash-системы, объединяющей в одно целое много сложных компонентов, причем все эти компоненты доступны вам через одно и то же окно просмотра (browser) и их можно настроить по вашему вкусу. Размещение вашего приложения отдельно от Интернета (offline), также может явиться изящным решением соответствующих проблем.

Если вы, например, пишете сайт на основе Flash для компании, торгующей компьютерным оборудованием, было бы очень удобно иметь графическую закладку на краю страницы, позволяющую пользователю одним нажатием кнопки перейти к дискуссионному форуму, посвященному интересующей его компьютерной детали, где можно было бы узнать мнение других покупателей. При этом пользователь фактически остается на той же странице, где описана деталь. Такие методы организации информации будут вскоре использоваться во многих Интернет-приложениях. Вы также можете присоединиться и создавать аналогичные приложения на основе среды MX.

Тот же самый дискуссионный форум может взаимодействовать с другим компонентом, таким как программа интерактивного общения (live chat), администратор которой может получить ваше недавнее сообщение в форуме и послать вам сообщение, приглашающее к личному интерактивному разговору[1] по поводу вашей технической проблемы. В течение нескольких минут вы можете получить помощь либо в дискуссионном форуме, либо от администратора программы интерактивного общения. Тот же самый администратор может даже послать вам через Flash MX схематические диаграммы в формате JPEG.

Это только несколько примеров возможного использования Flash в качестве интегрированного решения ваших проблем. В случае Flash вам также не нужно беспокоиться о типе программы просмотра (browser) или операционной системы на стороне пользователя (в отличие от решений на основе HTML) - вы уже знаете, что Flash клиенты одинаковы на разных платформах.

Все это довольно труднодостижимо в системах, основанных на HTML. В отличие от Flash, вам нужно будет писать сценарии идентификации клиента, решать проблемы со вложенными таблицами стилей (CSS), возможными JavaScript несовместимостями, таблицами и т. д.

1. Другими словами, пригласит вас обсудить вашу проблему в приватном чате. - *Примеч. науч. ред.*

Как работает наш дискуссионный форум

В отличие от Flash 5, среда Flash MX позволяет более презентабельно представить наши данные, что очень полезно для дискуссионного форума. В этой среде также очень быстро проходит синтаксический разбор XML-документов. Для отображения потоков (threads) используется компонент ScrollPane (панель прокрутки, поставляемая вместе со средой Flash MX) и форматирование текста TextField (с помощью ActionScript).

В нашем форуме также используется технология .NET. Сервер получает команды и данные от клиента, обрабатывает данные и возвращает результаты. Это сложная система, но она построена таким образом, что может использоваться во многих разных приложениях и обрабатывать многие типы запросов.

Ниже приведено подробное описание того, как работает сервер (backend) нашего форума. В этом описании довольно много технических деталей, оно поможет вам лучше понять идеи, использованные при написании сервера. После этого мы разберем подробности работы Flash-клиента.

Описание процесса работы сервера

В двух первых главах в качестве языка программирования на сервере в основном рассматривался ColdFusion. Хотя прилагался также соответствующий код на ASP.NET и Java, дизайн этого кода не обсуждался. Одной из задач этой книги является обучение тому, как один и тот же код можно повторно использовать в разных проектах как на стороне клиента, так и на стороне сервера. Вы увидите, что многие классы используются многократно во всех проектах этой книги. Так как наша книга посвящена не объектно-ориентированному программированию, мы рассмотрим соответствующие понятия лишь вкратце.

Ключевой составляющей многократно используемого кода является объект TransactionController (контроллер транзакций). Он используется как в C#, так и в Java для перенаправления запросов на выполнение. Работа программы разбивается на несколько транзакций. В большинстве проектов, например, есть транзакция Login (войти в систему) или CreateUser (создать пользователя). Обычно транзакции должны выполняться в определенном порядке. Flash-клиент посылает серверу запрос на транзакцию при помощи XML и HTTP, и сервер отвечает после завершения обработки транзакции.

Ниже описываются по шагам основные составляющие процесса работы сервера.

1. Клиент (Flash MX-дискуссионный форум в формате SWF) посылает по HTTP на сервер (страничке TransactionController.aspx) XML-документ стандартного формата. Это тот же самый основной XML-формат, который используется во всех проектах данной книги. Применение стандартного формата значительно облегчает многократное использование кода. Код серверной странички, упомянутой выше, находится в файле TransactionController.cs. Этот файл находится на CD-ROM-диске в C# (C-Sharp)-ыразделе.

2. Контроллер транзакций проводит синтаксический разбор XML-документа и извлекает имя транзакции в виде строки (например, GetTopics, GetThread, CreateUser ...). Затем это имя и оставшаяся часть XML-документа помещаются в новый объект TransInfo.
3. TransactionController создает объект TransactionService и передает ему объект Translnfo.
4. Объект TransactionService извлекает имя транзакции из объекта Translnfo и динамически создает копию соответствующего класса транзакции в виде объекта типа object. Затем осуществляется приведение типа только что созданного объекта к типу ITransaction. Класс ITransaction - это класс-интерфейс, реализуемый всеми транзакциями (GetTopics, GetThread, CreateUser ...).
5. TransactionService вызывает метод Execute созданного объекта транзакции. Объект транзакции выполняет свою работу на основе данных, содержащихся в объекте Translnfo. Затем он собирает объект TransResult и возвращает его объекту TransactionService.
6. TransactionService возвращает объект TransResult контроллеру TransactionController.
7. TransactionController извлекает XML-документ из переданного ему объекта TransResult и возвращает этот документ клиенту (дискуссионному форуму Flash MX).

Это довольно детальное объяснение процесса, показывающее все его возможности. Процесс может обрабатывать любые типы запросов, он также является устойчивым, так как он может перехватить и обработать любые возможные ошибки. Чтобы сделать весь процесс немного более понятным, на рис. 3.2 приведена схема процесса работы программы.

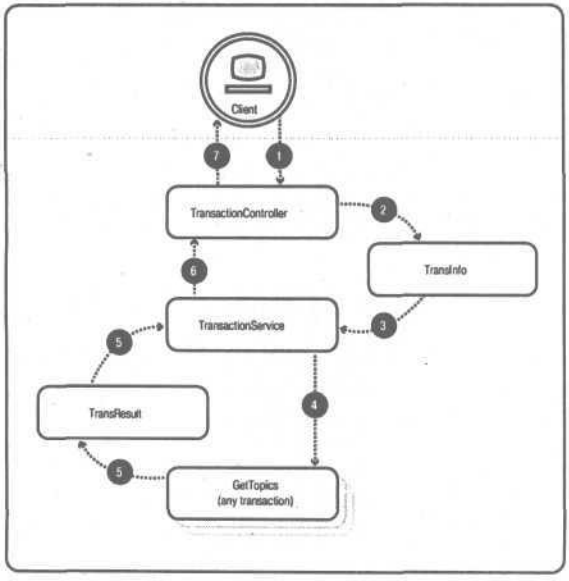

Рис. 3.2. Схема работы сервера (backend)

Если вы пишете на C# или интересуетесь файлами этого языка (.cs), вы всегда можете запустить ваш любимый редактор и изучить их во всех деталях. Если вы захотите внести какие-нибудь изменения, на CD-ROM-диске есть также командный файл компиляции (compileCSharp.bat в директории chapter_3/C-Sharp), с помощью которого можно перекомпилировать ваш .dll-файл.

Помните о том, что файл Transactions.cs является основой всей нашей системы. Он находится на сервере и написан полностью в стиле объектно-ориентированного программирования. Код из этого файла умеет обрабатывать запросы разных типов и используется во всех приложениях, описанных в данной книге (в их вариантах, основанных на .NET технологии). При желании вы можете еще больше расширить возможности этого кода. Он работает как с Access, так и с SQL, код программы одинаков в обоих случаях. Все необходимые дополнительные детали задаются во вспомогательном файле.

Сбор данных

Мы используем базу данных Access (.MDB). В этой базе данных (рис. 3.3) содержится вся информация (данные о пользователе, темы, потоки (threads) и сообщения), необходимая для работы дискуссионного форума. Таблица под названием Messages связывает между собой остальные таблицы в схеме базы данных. На CD-ROM-диске в директории гл. 3 приведен образец файла базы данных (MessageBoard.mdb). Этот образец содержит данные, созданные на этапе тестирования форума.

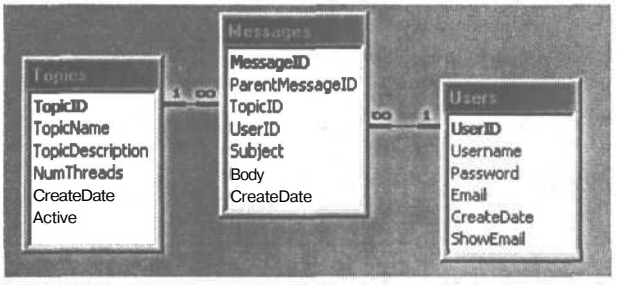

Рис. 3.3. Таблицы базы данных Access и их взаимосвязь

Интерфейс пользователя: Flash-клиент

Скопируйте файл MB.fla с CD-ROM-диска (из директории гл. 3) и откройте сделанную копию; на рабочем поле у вас должен появиться дискуссионный форум, в общем виде. Обратите внимание на выведенные наверх метки (labels), указывающие на различные функции программы.

Загрузка, первый кадр

Давайте начнем со слоя Actions (действия) (рис. 3.4). В первом кадре этого слоя помеченном Load (загрузка) вы найдете следующий ActionScript-код (являющийся частью процесса загрузки):

```
// Включение кода объекта ServerData
#include "ServerData.as"
// Включение общих функций на тему транзакций
#include "CommonTransactionFunctions.as"
// Включение кода панели загрузки
#include "LoadingPanel.as"
// Включение кода панели, сообщающей об ошибке
#include "ErrorPanel.as"
// Включение кода панели сообщений
#include "MessagePanel.as"
// Глобальные переменные
    global.loadingPanelDepth = 100;
    global.errorPanelDepth = 101;
    global.messagePanelDepth = 102;
    global.clipDepth = 1000;
    global.topicsLoaded = false;
// Конфигурация объекта ServerData
sd = new ServerData();
sd.setMethod(sd.SEND_AND_LOAD);
sd.setLanguage(sd.COLD_FUSION);
sd.setURL("http://www.electrotank.com:8000/book/cf/mb/Controller.cfm");
// Подключение загрузочной панели
root.attachMovie("LoadingPanel", "lPanel",loadingPanelDepth);
// Конфигурация панели
lPanel.setTitle("Loading ServerSide MX Board");
lPanel.setMessage("0% loaded");
lPanel.centerClip();
// Получение размера клипа
totalFileSize = _root.getBytesTotal();
// Начало события onEnterFrame (данная функция сообщает о состоянии загрузки)
root.onEnterFrame = function() {
    // Получим размер загруженных данных
    currentFileSize = _root.getBytesLoaded();
    // Проверим состояние дел с загрузкой
    if(currentFileSize < totalFileSize) {
        // Вычислим процент полученных данных
        percent = (currentFileSize / totalFileSize) * 100;
        percent = Math.round(percent);
        // Обновим сообщение о текущем состоянии загрузки
        lPanel.setMessage(percent + "% loaded");
} else {
```

```
            // Удалим загрузочную панель
            lPanel.unloadMovie();
            // Удалим данное событие
            _root.onEnterFrame = null;
            // Продолжение выполнения
            _root.nextFrame();
        }
    } // Конец события onEnterFrame
    // Инициализация звука щелчка кнопкой мыши
    click = new Sound();
    click.attachSound("click");
    click.setVolume(50);
    // Остановим клип
    stop();
```

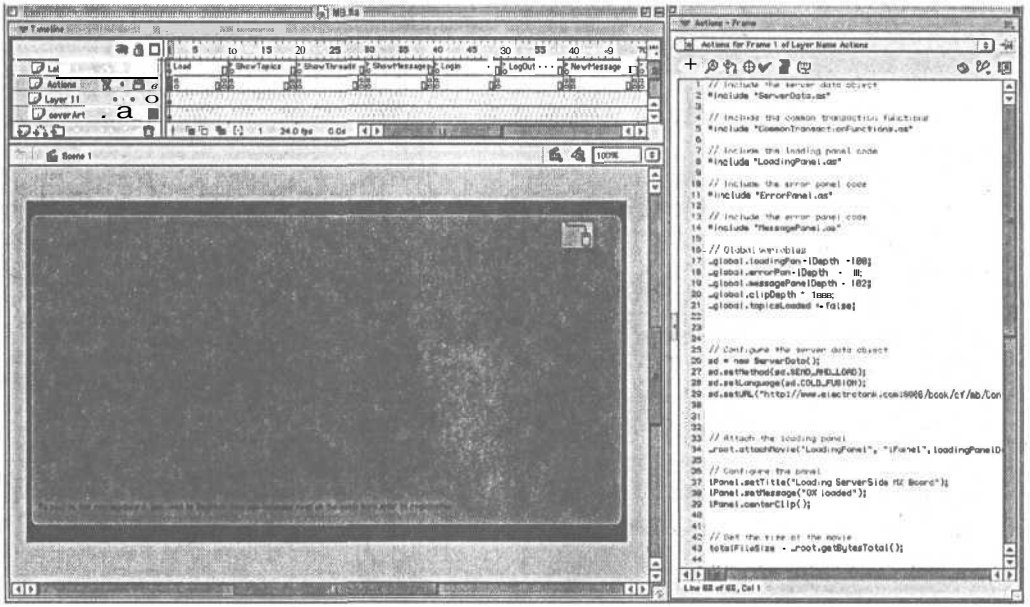

Рис. 3.4. Первый кадр слоя Actions содержит код, загружающий сам SWF-компонент

В этом коде вы видите, как наша программа обретает форму. В начале включается (include) несколько ActionScript-файлов, что обеспечивает импорт основного библиотечного кода программы при компиляции и публикации SWF. Вы можете самостоятельно попробовать разобраться в этом библиотечном коде, его объем слишком велик и не позволяет привести разъяснения в рамках этой книги. Большое количество комментариев должно значительно облегчить его понимание.

Потом определяются глобальные переменные (новая особенность среды Flash MX-переменные, существующие на глобальном уровне и доступные отовсюду), задающие координаты элементов интерфейса, далее еще одной глобальной переменной topicsLoaded присваивается значение false (темы еще не загружены).

Затем задается модель данных сервера, язык сервера и URL соответствующего управляющего файла Controller.cfm.

Далее конфигурируется загрузочная панель (для последующего показа статуса загрузки SWF). Следующие несколько строчек обеспечивают загрузочную панель (lPanel) необходимой информацией. Как только SWF загружен, панель прекращает работу (переменной onEnterFrame присваивается значение null). После окончания загрузки SWF-данных клип переходит к следующему кадру, покидая onEnterFrame цикл загрузочной панели. Здесь также создается звуковой объект (click), который будет использоваться в панели прокрутки тем, для реакции на движение мыши поверх панели (rollover feedback).

Загрузка, второй кадр

Во втором кадре слоя Actions находится код, загружающий данные самого дискуссионного форума.

```
// Включаем класс PropertiesManager
#include "PropertiesManager.as"
pm = new PropertiesManager();
pm.load("MBProperties.xml");
pm.onLoad = propertiesLoaded;
// Подключение загрузочной панели
    root.attachMovie("LoadingPanel", "lPanelProps", loadingPanelDepth);
// Конфигурирование панели
lPanelProps.setTitle("Loading Properties");
lPanelProps.setMessage("0% loaded");
lPanelProps.centerClip();
// Получение размера файла с атрибутами (properties)
totalPropsSize = pm.getBytesTotal();
// Начало события onEnterFrame (проверка текущего состояния загрузки)
root.onEnterFrame = function() {
    // Получим размер загруженных данных
    currentPropsSize = pm.getBytesLoaded();
    // Проверим состояние дел с загрузкой
    if(currentPropsSize < totalPropsSize) {
        // Вычислим процент полученных данных
        percent = (currentPropsSize / totalPropsSize) * 100;
        percent = Math.round(percent);
        // Обновим сообщение о текущем состоянии загрузки
        lPanelProps.setMessage(percent + "% loaded");
    } else {
        // Удалим панель
        lPanelProps.unloadMovie();
        // Удалим данное событие
        _root.onEnterFrame = null;
        // Продолжение выполнения
        _root.nextFrame();
```

```
    }
} // конец события onEnterFrame
function propertiesLoaded(success) {
    // Проверим результат загрузки
    if(success) {
        // Получим данные
        _global._serverLanguage = this.getProperty("ServerLanguage");
        _global._controllerURL = this.getProperty("ControllerURL");
    // Перейдем к метке ShowTopics
    gotoAndPlay("ShowTopics");
  } else {
    // Подключим панель, сообщающую об ошибке
    root.attachMovie("ErrorPanel", "propsErrorPanel", errorPanelDepth);
    // Конфигурирование панели
    propsErrorPanel.setMessage("Error Loading Properties");
    propsErrorPanel.centerClip();
  }
}
// Остановим клип
stop();
```

Сначала загружается код из файла PropertiesManager.as, потом динамически создается панель загрузки. После загрузки XML-данных ActionScript проверяет результат загрузки. В случае ошибки создается панель, сообщающая об ошибке при загрузке атрибутов. В случае успеха загрузочная панель удаляется.

Ниже приводится содержание XML-документа MBProperties.xml (загружаемого в предыдущем коде):

```
<Properties>
  <Property>
    <Name>ServerLanguage</Name>
    <Value>ASPNET</Value>
  </Property>
  <Property>
    <Name>ControllerURL</Name>
    <Value>http://msgboard.flashbook.hostingdot.net/TransactionController.aspx/</Value>
  </Property>
</Properties>
```

Здесь объявляется язык сервера и задается URL файла с кодом контроллера, для ассоциации с соответствующими методами обработки.

В случае успешной загрузки программа переходит к кадру, помеченному меткой ShowTopics. Так как данные из XML файла MBProperties.xml уже загружены, можно начать загрузку тем. Весь код, описанный выше, выполняется очень быстро благодаря способности среды Flash MX быстро обрабатывать XML-данные. К тому же файлы .NET заранее скомпилированы, что также ускоряет обработку требуемых данных.

Демонстрация тем, десятый кадр

В десятом кадре на основной монтажной линейке (рис. 3.5) вы видите, что наша программа уже близка по виду к обычному форуму. Сверху находятся кнопки для перехода в окна Home (домой), Register (зарегистрироваться) и Login (войти в систему). В данном окне пользователи имеют доступ к темам, потокам (threads) и ответам на сообщения, но не могут поместить свои сообщения. Для активного участия в форуме им нужно зарегистрироваться и войти в систему.

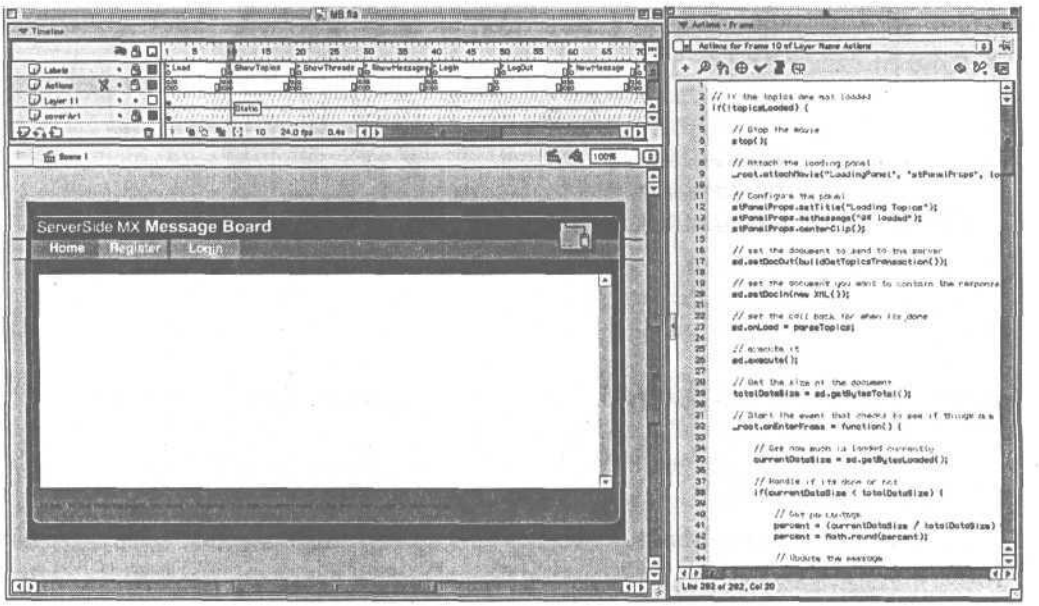

Рис. 3.5. Данные о темах демонстрируются в десятом кадре

На рабочем поле находится копия (MC instance) панели прокрутки (ScrollPane) под названием topicPane (рис. 3.6). У этой копии установлены следующие параметры:

Horizontal Scroll = false (отменена вертикальная прокрутка);

Vertical Scroll = auto (автоматическая вертикальная прокрутка; прокрутка появляется по необходимости, когда содержимое выходит за пределы панели; в случае 'true' прокрутка присутствует всегда);

Drag Content = false (панель нельзя перетаскивать).

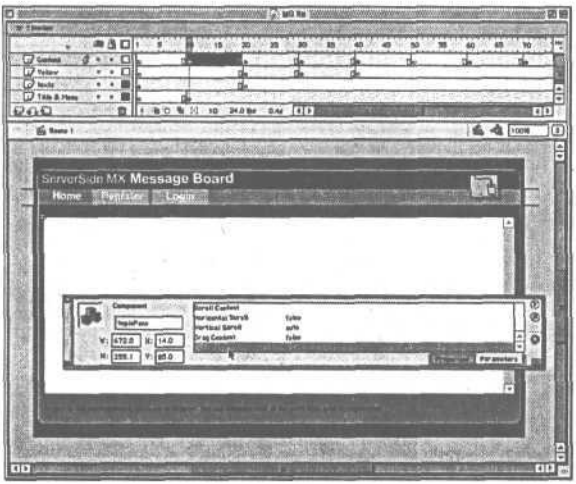

Рис. 3.6. Компонент ScrollPane, содержащий все данные о темах

Код на последующих стадиях монтажной линейки очень похож на код, обсуждаемый в данном разделе. Этот код наглядно демонстрирует возможности Flash MX по размещению компонентов и форматированию текста.

Помещение в панель прокрутки (ScrollPane) сразу нескольких компонентов

Данная панель прокрутки может прокручивать только один компонент, например загруженное изображение в формате JPEG или анимационный клип. Наша программа решает проблему прокрутки сразу нескольких объектов путем динамического создания одного клипа, помещения этого клипа в панель прокрутки и последующего его заполнения дочерними клипами (с текстами тем).

Обратите внимание на следующий код из десятого кадра (этот код довольно велик по объему):

```
// Если темы еще не загружены
if(!topicsLoaded) {
    // Остановим клип
    stop();
    // Подключим панель загрузки
    _root.attachMovie("LoadingPanel", "stPanelProps", loadingPanelDepth);
// Конфигурация панели
stPanelProps.setTitle("Loading Topics");
stPanelProps.setMessage("0% loaded");
stPanelProps.centerClip();
// Зададим документ, который будет послан на сервер
sd.setDocOut(buildGetTopicsTransaction());
// Зададим документ, который будет содержать ответ
sd.setDocIn(new XML());
// Зададим функцию (call back) для события onLoad (после загрузки)
sd.onLoad = parseTopics;
```

```
// Выполним это
sd.execute();
// Получим размер документа
totalDataSize = sd.getBytesTotal();
// Начало события onEnterFrame (проверка состояния загрузки)
_root.onEnterFrame = function() {
    // Получим размер загруженных данных
    currentDataSize = sd.getBytesLoaded();
    // Проверим состояние дел с загрузкой
    if(currentDataSize < totalDataSize) {
        // Вычислим процент полученных данных
        percent = (currentDataSize / totalDataSize) * 100;
        percent = Math.round(percent);
        // Обновим сообщение о текущем состоянии загрузки
        stPanelProps.setMessage(percent + "% loaded");
    } else {
        // Удалим данное событие
        _root.onEnterFrame = null;
        // Продолжение выполнения
        _root.nextFrame();
        }
    } // Конец события onLoad
} else {
    // Отобразим темы на экране
    showTopics();
} // Конец if(!topicsLoaded)
// Функция parseTopics
// Эта функция обрабатывает полученный XML-документ
// и создает клипы тем
function parseTopics() {
    // Получим данные от объекта ServerData
    var dataIn = this.getDocIn();
    // Проверим успешность завершения транзакции
    if(!wasSuccessful(dataIn)) {
        // Подключим панель, демонстрирующую сообщение об ошибке
        _root.attachMovie("ErrorPanel", "topicsErrorPanel",
errorPanelDepth); // продолжение строки
        // Конфигурация панели
        topicsErrorPanel.setMessage(errorMessage);
        topicsErrorPanel.centerClip();
            return;
    } // Конец if(!wasSuccessful(dataIn))
    // Создаем клип, который будет содержать темы
    _root.createEmptyMovieClip("topics", clipDepth++);
dataNode = findDataNode(dataIn);
topicsNode = dataNode.firstChild;
```

```
topicNodes = topicsNode.childNodes;
top=0;
width = 660;
for(i = 0; i < topicNodes.length; i++) {
    topicNode = topicNodes[i];
    topicChildren = topicNode.childNodes;
    id = topicNode.attributes.ID;
    name = topicChildren[0].firstChild.nodeValue;
    description = topicChildren[1].firstChild.nodeValue;
    numThreads = topicChildren[2].firstChild.nodeValue;
    createDate = topicChildren[3].firstChild.nodeValue;
    backColor= (i % 2)? 0xE9E9E9 : OxD9D9D9;
tempTopic = createTopic(_root.topics, i, width, backColor,
id, name, description);//продолжение строки
        tempTopic._y= Math.round(top);// добавлено округление
        top += Math.round(tempTopic._height);// добавлено округление
    }
    topicPane.setScrollContent(topics);
    //topicPane._visible = true;
    // Удалим панель
    stPanelProps.unloadMovie();
    // Темы загружены
    topicsLoaded = true;
}.// Конец функции parseTopics
//Функция buildGetTopics
// Эта функция создает транзакцию 'GetTopics'
function buildGetTopicsTransaction() {
    // Создаем транзакцию 'GetTopics'
    getTopicsRequest = buildBaseRequest("GetTopics");
    // Возвращаем XML-объект
    return  getTopicsRequest;
} // Конец функции buildGetTopics
// Функция createTopic
// Эта функция добавляет в клип тем одну тему
function createTopic(container, depth, width, color, id, title,
description) {// продолжение строки
    name = "topic_" + depth;
    container.createEmptyMovieClip(name, depth);
    topic = container[name];
    topic.id = id;
    topic.createTextField("title", 2, 0, 0, width, 0);
    topic.title.text = title;
    with(topic.title) {
        multiline = true;
        wordwrap = true;
        border = false;
```

```
        autoSize = "center"; // было center
        selectable = false;
        embedFonts = true;
}
topic.titleformat = new TextFormat();
with(topic.titleFormat) {
color = 0xcc0000; // красный (deep red)
font = "Genetica";
bold = false;
leftMargin = 3; // проверка
size = 10;
}
topic.title.setTextFormat(topic.titleformat);
    yPos = topic.title._height + topic.title._x;
topic.createTextField("description", 5, 0, yPos, width, 0);
// было "description", 3, 0, yPos, width, 0);
topic.description.text = description;
with(topic.description) {
    multiline = true;
    wordwrap = true;
    border = false;
    autoSize = "left"; // было center
    selectable = false;
    embedFonts = true;
}
topic.descriptionformat = new TextFormat();
with(topic.descriptionformat) {
    color = 0x000000; // черный
    font = "Standard";
    bold = false;
    leftMargin = 3; // проверка
    size = 8;
}
topic.description.setTextFormat(topic.descriptionformat);
    height = Math.round(topic._height);
    height = Math.round(topic._height);
    topic.createEmptyMovieClip("background", 1);
    topic.background.color = color;
    with(topic.background) {
        beginFill(color);
        lineTo(width, 0);
        lineTo(width, height);
        lineTo(0, height);
        endFill();
}
// Функция-обработчик события onRelease
```

```
topic.background.onRelease = function() {
        // Скроем все, относящееся только к окну тем
        _root.hideTopics();
        // Зададим идентификатор темы
        _root.topicID=this._parent.id;
        // Зададим заголовок темы
        _root.topicTitle = this._parent.title.text;
        // Воспроизведем звук щелчка мыши (click)
      _root.click.start(0,0);
        // Перейдем к метке ShowThreads (показать потоки)
      _root.gotoAndPlay("ShowThreads");
}
// Инициализируем эти события перед возвращением темы (так как topic временная переменная)
topic.background.onRollOver = function(){
      _root.click.start(0,0);
      this._alpha = 30;
}
topic.background.onRollOut = function(){
      this._alpha =100;
   }
// Возвратим только что созданную тему
return topic;
} // Конец функции createTopic
// Функция showTopics
// Эта функция снова показывает уже загруженные темы
function showTopics() {
   // Сделаем клип тем снова видимым
   topics._visible = true;
   // Зададим содержимое панели прокрутки
   topicPane.setScrollContent(topics);
} // Конец функции showTopics
// Функция hideTopics
// Эта функция скрывает панель тем
function hideTopics() {
   // Скроем клип тем
   topics._visible = false;
} // Конец функции hideTopics
```

Так как в этом ActionScript-коде очень много комментариев, вы, наверно, уже понимаете, что происходит, и я не буду описывать все подробности. Вы видите уже знакомый процесс загрузки, а также функцию для синтаксического разбора XML-документа с темами.

Найдите в коде строчку со следующим комментарием: // Создаем клип, который будет содержать темы. В этом месте в полную силу используются новые возможности форматирования в среде Flash MX. В последующих строчках, с помощью

только ActionScript, создается пустой клип, экземпляр которого называется "topics".

"Подождите, но этот клип не связан с панелью прокрутки", — можете вы подумать. Мы позаботимся об этом немного позже. Создается пустой клип, наполняется содержимым (клипами тем), и потом связывается с компонентом прокрутки. Вы увидите, как все это происходит, через несколько минут.

Далее обрабатываются XML-данные и определяются длины заголовков тем, затем на основе этих заголовков создаются текстовые поля, расположенные друг над другом, с чередующимся цветом фона. Для большего понимания приведем соответствующий код.

```
backColor = (I % 2)? 0xE9E9E9 : OxD9D9D9;
tempTopic = createTopic(_root.topics, i, width, backColor, id, name, description);
//продолжение строки
tempTopic._y = Math.round(top);// добавлено округление
top += Math.round(tempTopic._height);//добавлено округление
```

Округление было добавлено в связи с тем, что программа использует пикселовые шрифты и для поддержания качества изображения координаты вложенных клипов должны быть целыми числами. Я считаю, что не сглаженные (non-anti-aliased) шрифты (например, обычные HTML-шрифты) все-таки удобней для чтения. У каждого разработчика свой стиль и приемы размещения динамических данных в приложениях, но использование пиксельных шрифтов (хорошие можно купить, например, на сайте www.miniml.com) может оказаться очень удобным для ваших пользователей.

i Форматирование текстовых объектов очень сильно помогает в работе

При планировании дискуссионного форума невозможно определить заранее точные размеры динамических текстовых объектов и их расположение друг относительно друга. В среде Flash 5 приходилось размещать текстовые поля вручную, либо пользоваться функцией attachMovieClip для размещения текстовых данных в более-менее презентабельном виде. Никогда не известно заранее, какое количество текста будет помещено в некоторые поля, поэтому приходилось полагаться на приблизительные оценки и надеяться на лучшее. В зависимости от количества внесенного текста иногда текстовые поля располагались хорошо, а иногда очень плохо. Теперь, в среде Flash MX, вы можете научить программу подстраиваться под пользователя - вы увидите, что темы расположены приятным для глаза образом, друг под другом, даже если одна из них занимает три строчки. С текстовыми объектами теперь можно обращаться как с клипами, что дает много возможностей для их настройки. Программа может работать с записями любых размеров, поддерживая при этом профессиональный презентабельный вид. В этом случае поведение Flash похоже на HTML-таблицы, только более управляемые.

Форматирование текста в текстовом поле Title (заголовок) выполняется при помощи следующего ActionScript-кода:

```
topic.titleformat = new TextFormat();
```

```
    with(topic.titleFormat) {
        color = 0xcc0000; // красный (deep red)
        font = "Genetica";
        bold = false;
        leftMargin = 3; // проверка
        size = 10;
    }
    topic.title.setTextFormat(topic.titleformat);
```

Объявленный шрифт, Genetica, на самом деле является именем связи (Linkage name) с символом шрифта (Genetica), созданным в библиотеке (Library). Таким образом с динамически созданным текстовым полем можно ассоциировать шрифт для отображения текста.

Создание в ActionScript символа шрифта

Чтобы создать в библиотеке (Library) новый символ шрифта, из меню Library выберите New Font; вы увидите диалоговое окно, в котором можно задать имя символа, выбрать шрифты, доступные в вашей системе, и задать атрибут Bold (полужирный) или Italic (курсив). Имя, данное символу фонта, само по себе еще не является именем связи (Linkage name) - это просто имя символа. Когда вы закончили с созданием нового символа шрифта в меню Library, нажмите правой кнопкой мыши поверх символа в библиотеке и выберите Linkage. В этом меню вы можете дать символу имя связи (Linkage name) и выбрать возможные типы экспорта: для ActionScript (в первом кадре или в момент первого появления), для возможного использования при работе программы (runtime sharing) и т. д. Имя связи позволяет ссылаться на данный символ и использовать его из ActionScript.

В среде Flash 5 вам нужно было бы заранее задать шрифт текстового поля (статического, динамического или поля ввода). Среда Flash MX позволяет динамически связать шрифт с текстовым полем, созданным в ActionScript, как и сделано в нашей программе. Вы можете задать таким образом несколько шрифтов, в соответствии с идеями объектно-ориентированного программирования. Темы запрограммированы именно таким образом. Ниже приведен код, форматирующий текст описаний (description) заголовков тем.

```
with(topic.descriptionformat) {
    color = 0x000000; // черный
    font = "Standard";
    bold = false;
    leftMargin = 3; // проверка
    size = 8;
}
topic.description.setTextFormat(topic.descriptionformat);
```

Имя связи "Standard" на самом деле указывает на шрифт Standard 07_53 (рис. 3.7).

Использование одного шрифта с разными атрибутами

Если вы хотите употребить в пределах одного динамически созданного текстового поля один и тот же шрифт с разными атрибутами (например, обычный и полужирный), HTML-форматирование (<h>) не сработает. В этом случае нужно пользоваться функцией set-TextFormat наряду с объектами формата.

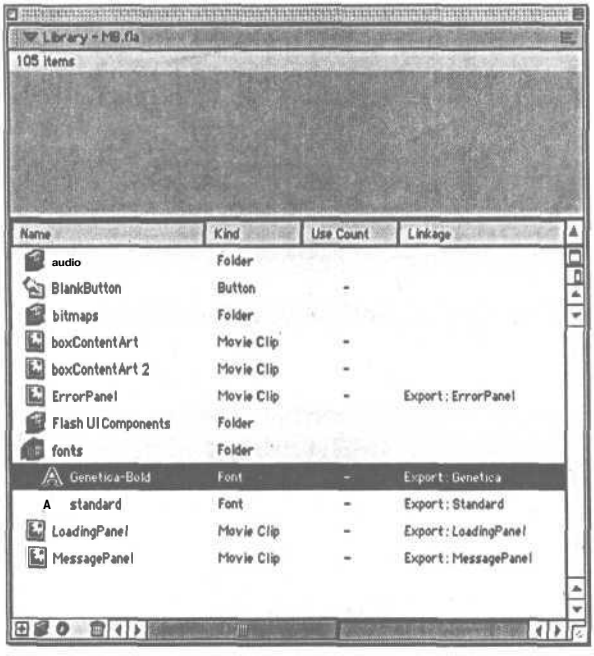

Рис. 3.7. Связь со шрифтом в библиотеке (Library). Шрифт Standard 07_53 связывается под именем Standard - совпадение имен необязательно

В среде Flash MX с клипами можно ассоциировать события-кнопки (движение мыши и нажатие кнопок мыши). При использовании программы вы увидите, что в окне тем программа реагирует на проведение мышью поверх заголовков (путем звука и изменения фона). Для каждого заголовка и описания фон задается путем динамического создания (drawing API) клипов с названием (копии) background. Реакция на событие rollOver (поверх) достигается путем связывания соответствующих событий с клипом фона:

```
topic.background.onRollOver = function(){
        _root.click.start(0,0);
        this._alpha = 30;
    }
    topic.background.onRollOut = function(){
        this._alpha = 100;
    }
}
```

В случае события rollOver воспроизводится заданный звук и атрибут фона _alpha уменьшается до 30 %. В случае rollOut атрибут _alpha опять становится равным 100%. Видите, насколько это ближе к методам ООР, нежели программирование в среде Flash 5. Для воспроизведения звука в среде Flash 5 нужно было бы поместить звуковое событие внутри функции события rollOver и вручную изменить значение _alpha. А в нашей программе в качестве кнопки используется клип - динамически созданный в ActionScript - и мы можем легко изменить любые атрибуты. Такие вещи в среде Flash 5 абсолютно невозможны, и они значительно расширяют возможности программиста.

Давайте повторим еще раз, к каждой копии клипа фона мы добавляем функции, срабатывающие в случае событий-кнопок. Фирма Macromedia, добавив такую возможность, определенно сделала шаг в правильном направлении (по сравнению с Flash 5). Теперь вы можете создавать клипы, которые можно менять и которые при этом могут служить в качестве кнопок. Это еще один пример того, как Flash становится все более объектно-ориентированным.

При создании отдельных клипов для заголовков тем родительский клип topics остается невидимым; после создания всех дочерних клипов клип topics переводится в видимое состояние. Это осуществляется с помощью двух функций, по одной на каждое состояние видимости:

```
function showTopics() {
    // Сделаем клип topics видимым
    topics._visible = true;
    // Зададим содержимое панели прокрутки
    topicPane.setScrollContent(topics);
} // Конец функции showTopics
// Функция hideTopics
// Эта функция скрывает клип topics
function hideTopics() {
    // Сделаем клип topics невидимым
    topics._visible = false;
} // Конец hideTopics
```

Следующая строчка ActionScript связывает клип с панелью прокрутки (с образцом под названием topicPane):

```
topicPane.setScrollContent(topics);
```

Таким образом, набор заголовков тем, содержащийся внутри клипа topics, может теперь прокручиваться с помощью компонента ScrollPane. Помните о том, что ScrollPane может прокручивать только один объект, SWF, клип или JPEG. В нашем коде много дочерних клипов вкладываются в единственный родительский клип, который, в свою очередь, прокручивается по необходимости.

ShowThreads (показать потоки), ShowMessages (показать сообщения), кадры 20 и 30

Реализации показа тем, потоков и сообщений (рис. 3.8 и рис. 3.9) очень сильно похожи. Вы можете взглянуть на соответствующий ActionScript код в кадрах 20 и 30. Вы увидите, что этот код очень сильно похож на код из кадра 10 (ShowTopics). Для отображения данных также используется панель прокрутки и форматирование текста.

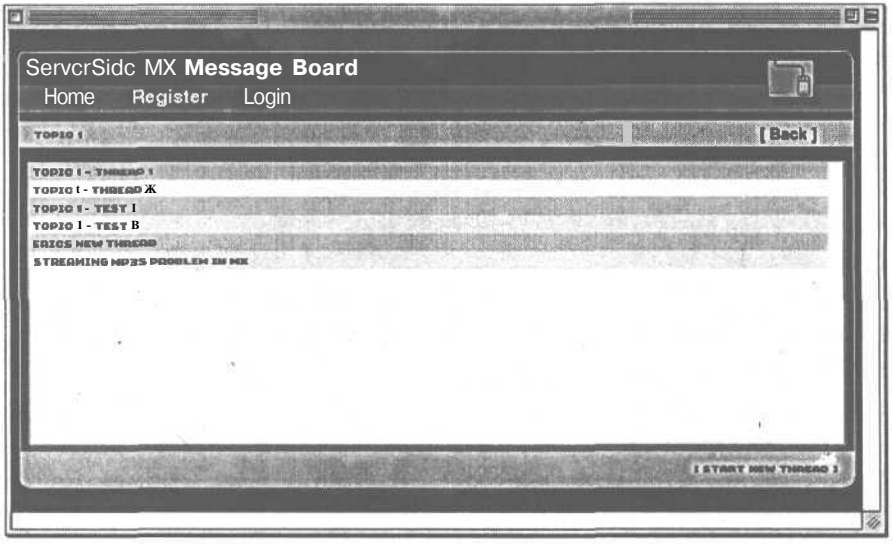

Рис. 3.8. Пользователь может выбрать поток (thread)

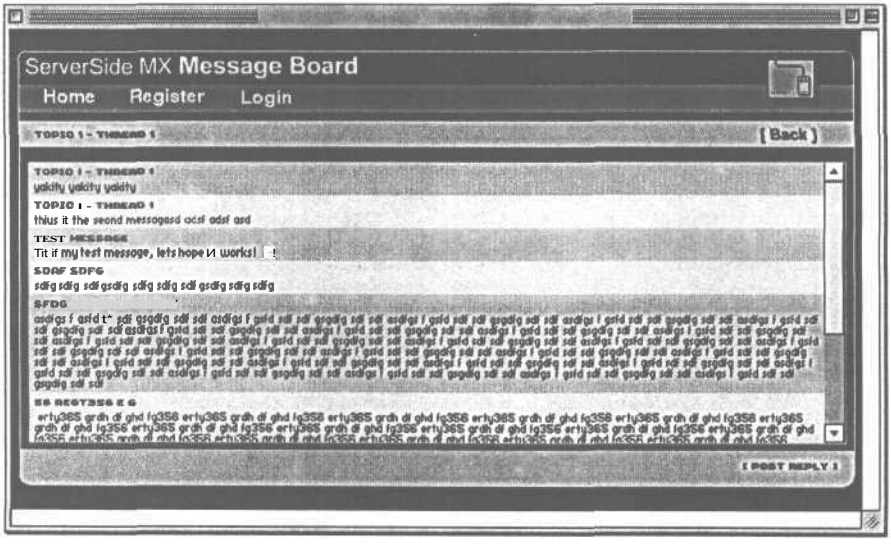

Рис. 3.9. Код, отображающий потоки и сообщения, очень сильно похож на код отображения тем

Давайте двинемся вперед в нашем FLA и перейдем к кадру 40, где пользователь может войти в систему (если он уже зарегистрирован в форуме). Попытка войти без предварительной регистрации заканчивается сообщением об ошибке (отображаемым системой).

Вход в систему (Login), кадр 40

В кадре 40 (Login) вы найдете на рабочем поле панель входа в систему, состоящую из двух полей ввода: Username (имя) и Password (пароль) (рис. 3.10). Обратите внимание, что текстовые поля не связаны напрямую с именами переменных, они связаны с их образцами (instances). Помните о том, что с текстовыми полями теперь можно обращаться как с клипами.

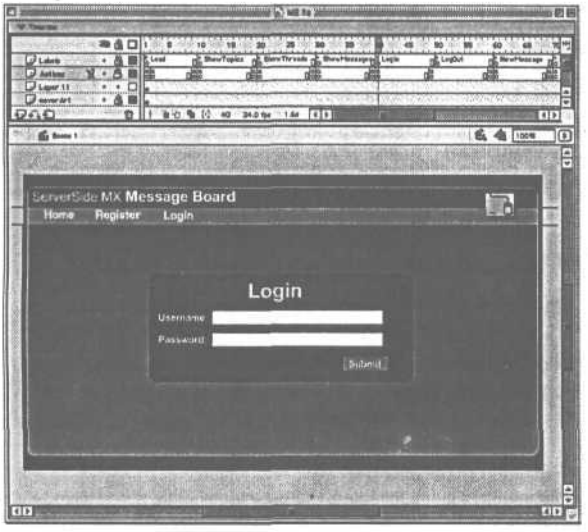

Рис. 3.10. Окно Login является панелью интерфейса с двумя текстовыми полями ввода

В кадре 40 определено несколько функций: loginUser(), parseLoginResults() и buildLoginTransaction(username, password).

Здесь также есть кнопка для обработки и отправки данных, с которой связан следующий код:

```
on(press) {
    // Отправка данных
    loginUser();
}
```

В самом кадре 40 находится код функции loginUser и ActionScript-код, проверяющий достоверность введенных данных. После успешного входа в систему пользователю сообщается об успехе и потом осуществляется переход в окно тем (ShowTopics). Если данные, введенные пользователем, неверны, выдается сообщение об ошибке. Ниже приведен код с комментариями.

```
// Функция loginUser
// Эта функция используется для входа в систему
function loginUser() {
```

```
        // Построим login транзакцию
        var loginRequest = buildLoginTransaction(username.text,
password.text); // продолжение строки
// Зададим документ, который будет послан на сервер
    sd.setDocOut(loginRequest);
    // Зададим документ, который будет содержать ответ
    sd.setDocIn(new XML());
    // Зададим функцию для события onLoad
    sd.onLoad = parseLoginResults;
    // Выполним это
    sd.execute();
} // Конец функции loginUser
// Функция parseLoginResults
// Эта функция используется для синтаксического разбора данных,
// полученных с сервера
function parseLoginResults() {
    // Получим данные от объекта ServerData
    var dataIn = this.getDocIn();
    // Проверим успешность завершения транзакции
    if(!wasSuccessful(dataIn)) {
        // Подключим панель для сообщения об ошибке
        _root.attachMovie("ErrorPanel", "loginErrorPanel",
errorPanelDepth);// продолжение строки
        // Конфигурация панели
        loginErrorPanel.setMessage(errorMessage);
        loginErrorPanel.centerClip();
        return;
} // конец if(!wasSuccessful(dataIn))
        // Подключим панель для сообщения об успехе
        _root.attachMovie("MessagePanel", "loginMessagePanel",
messagePanelDepth);// продолжение строки
        // Конфигурация панели
        loginMessagePanel.setMessage("You have logged in successfully.
Press 'OK' to continue");// продолжение строки
        loginMessagePanel.setButtonText("OK");
        loginMessagePanel.centerClip();
        // При закрытии панели перенаправим пользователя
        loginMessagePanel.messagePanelClose = function() {
        gotoAndPlay("ShowTopics");
    }
) // Конец функции parseLoginResults
// Функция buildLoginTransaction
// Эта функция содает XML-документ из данных пользователя
function buildLoginTransaction(username, password) {
        // Построим транзакцию 'Create User'
        var createUserRequest = buildBaseRequest("Login");
        // Получим узел данных
        var dataNode = findDataNode(createUserRequest);
        // Создание XML-кода для имени пользователя
        // Создадим элемент Username
        var usernameNode = createUserRequest.createElement("Username");
        // Создадим текстовый узел usernameValue
        var usernameValue = createUserRequest.createTextNode(username);
        // Добавим узел usernameValue к узлу usernameNode
        usernameNode.appendChild(usernameValue);
```

```
        // Добавим узел usernameNode к узлу данных dataNode
        dataNode.appendChild(usernameNode);
        // Создание XML-кода для пароля
        // Создадим элемент Password
        var passwordNode = createUserRequest.createElement("Password");
        // Создадим текстовый узел passwordValue
        var passwordValue = createUserRequest.createTextNode(password);
        // Добавим узел passwordValue к узлу passwordNode
        passwordNode.appendChild(passwordValue);
        // Добавим узел passwordNode к узлу данных dataNode
        dataNode.appendChild(passwordNode);
        // Вернем полученный XML-документ
        return createUserRequest;
} // Конец функции buildLoginTransaction
```

Новое сообщение (NewMessage), кадр 60

Данная часть программы (рис. 3.11) позволяет разместить новое сообщение; тело сообщения и тема (subject) сохраняются в базе данных. Если вы еще не вошли в систему, попытка начать новый поток (thread), добавить ответ или поместить новое сообщение приведет к сообщению об ошибке, напоминающему о том, что вы забыли войти в систему. Если вы еще не зарегистрированы, значит, вам нужно зарегистрироваться.

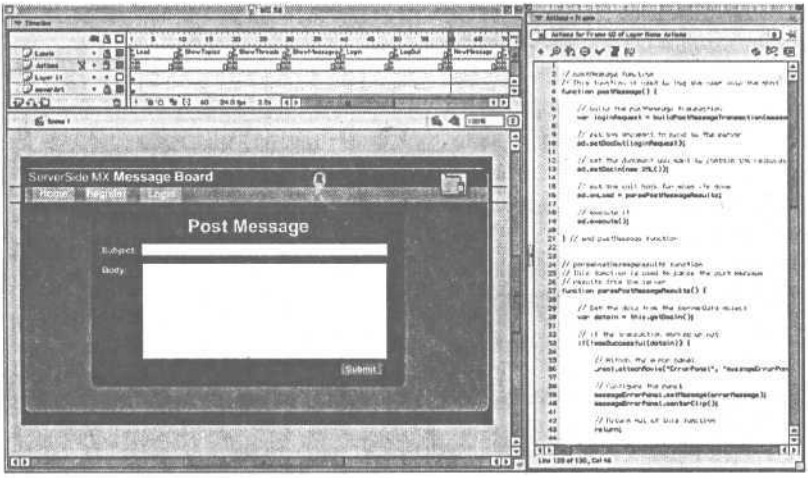

Рис. 3.11. Модуль NewMessage (новое сообщение)

Здесь определяется несколько функций: postMessage(), parsePostMessageResults() и buildPostMessageTransaction(subject, body). postMessage(),где является основной функцией, использующей обе оставшиеся функции, так что при вызове postMessage() выполняются все три.

Обратите особое внимание на комментарии в ActionScript-коде, так как они значительно облегчают понимание.

К кнопке отправки привязан следующий код:

```
on(press) {
    // Разместить сообщение
    postMessage();
}
```

Ниже приведен ActionScript-код, размещающий новое сообщение (кадр 60).

```
// Функция postMessage
// Эта функция используется для размещения сообщения
function postMessage() {
    // build the postMessage transaction
    var loginRequest = buildPostMessageTransaction(messageTitle.text,
    messageBody.text);//продолжение строки
// Зададим документ, который будет послан на сервер
    sd.setDocOut(loginRequest);
    // Зададим документ, который будет содержать ответ
    sd.setDocIn(new XML());
    // Зададим функцию для события onLoad
    sd.onLoad = parsePostMessageResults;
    // Выполним это
    sd.execute();
} // Конец функции postMessage
// Функция parsePostMessageResults
// Эта функция используется для обработки результатов,
// полученных с сервера
function parsePostMessageResults() {
    // Получим данные от объекта ServerData
    var dataIn = this.getDocIn();
    // Проверим успешность завершения транзакции
    if(!wasSuccessful(dataIn)) {
        // Подключим панель для сообщения об ошибке
        _root.attachMovie("ErrorPanel", "messageErrorPanel",
errorPanelDepth);//продолжение строки
        // Конфигурация панели
        messageErrorPanel.setMessage(errorMessage);
        messageErrorPanel.centerClip();
        return;
    } // конец if(!wasSuccessful(dataIn))
    // Подключим панель для сообщения об успехе
    _root.attachMovie("MessagePanel", "messageMessagePanel",
messagePanelDepth);// продолжение строки
    // Конфигурация панели
    messageMessagePanel.setMessage("Your message has been posted
successfully.\nPress 'OK' to continue");//продолжение строки
    messageMessagePanel.setButtonText("OK");
    messageMessagePanel.centerClip();
    // При закрытии панели перенаправим пользователя
```

```
        messageMessagePanel.messagePanelClose = function() {
        gotoAndPlay("ShowThreads");
        }
    } // Конец функции parsePostMessageResults
    // Функция buildPostMessageTransaction
        // Эта функция создает XML-документ с новым сообщением
        function buildPostMessageTransaction(subject, body) {
        // Построим транзакцию 'Post Message'
        var postMessageRequest = buildBaseRequest("PostMessage");
        // Получим узел данных
        var dataNode = findDataNode(postMessageRequest);
        // Создание XML-кода для заголовка сообщения (Subject)
        // Создадим элемент usernameNode
        var subjectNode = postMessageRequest.createElement("Subject");
        // Создадим текстовый узел subjectValue
        var subjectValue = postMessageRequest.createTextNode(subject);
        // Добавим узел subjectValue к узлу subjectNode
        subjectNode.appendChild(subjectValue);
        // Создание XML-кода для тела сообщения (Body)
        // Создадим элемент Body
        var bodyNode = postMessageRequest.createElement("Body");
        // Создадим текстовый узел bodyValue
        var bodyValue = postMessageRequest.createTextNode(body);
        // Добавим узел bodyValue к узлу bodyNode
        bodyNode.appendChild( bodyValue);
        // Создание XML-кода для сообщения (Message)
        // Создадим элемент messageNode
        var messageNode = postMessageRequest.createElement("Message");
        // Добавим атрибуты
        messageNode.attributes.TopicID = topicID;
        messageNode.attributes.ThreadID = threadID;
        // Добавим узел subjectNode к узлу messageNode
        messageNode.appendChild(subjectNode);
        // Добавим узел bodyNode к узлу messageNode
        messageNode.appendChild(bodyNode);
        // Добавим узел messageNode к узлу данных dataNode
        dataNode.appendChild(messageNode);
        // Вернем законченный XML-документ
        return postMessageRequest;
    } // Конец функции buildPostMessageTransaction
```

А сейчас вернитесь и освежите в памяти схему (backend) процесса работы сервера (рис. 3.2), приведенную в начале главы. Вы видите теперь, как обрабатывается транзакция postMessageRequest. Эта схема поможет вам понять, как обрабатываются разные типы запросов в нашей программе.

Регистрация, кадр 70

Для успешного входа в систему пользователю вначале необходимо зарегистрироваться (рис. 3.12) в дискуссионном форуме. Окно регистрации позволяет ввести имя пользователя и пароль, которые сохраняются в базе данных для последующего применения. При регистрации также требуется ввести электронный адрес и указать в соответствующем текстовом поле ввода, должен ли этот адрес быть видимым остальным пользователям. В данный момент в это текстовое поле можно ввести "Yes" или "No" - вы можете поменять это на более удобные компоненты типа селектора (radio buttons). Пользователи, не вошедшие в систему, получают сообщение об ошибке при попытке что-нибудь добавить (рис. 3.13).

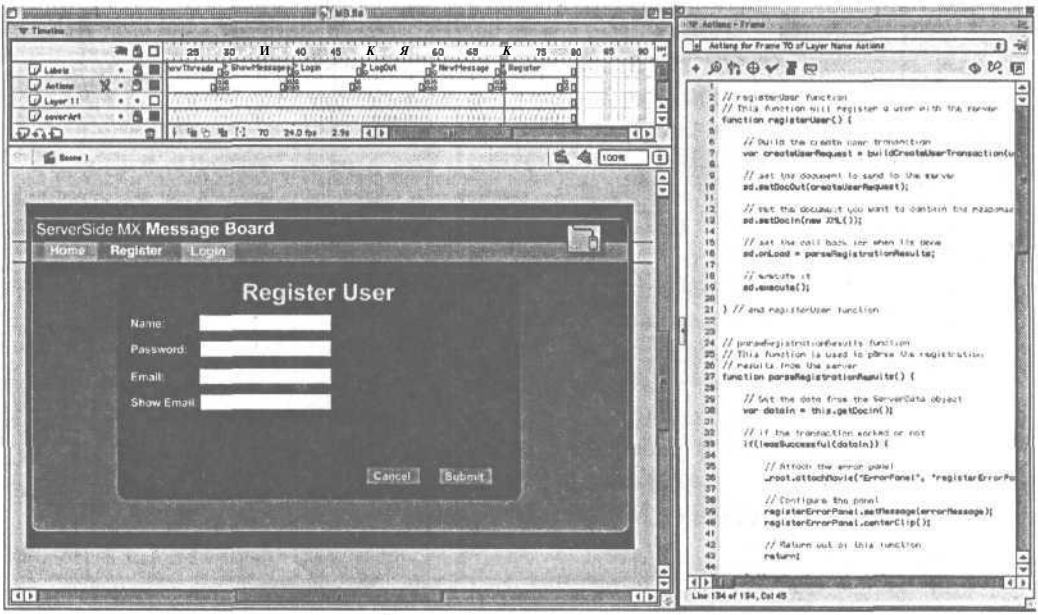

Рис. 3.12. Flash-форма собирает данные и регистрирует нового пользователя, давая возможность ему размещать новые потоки, сообщения и ответы на сообщения

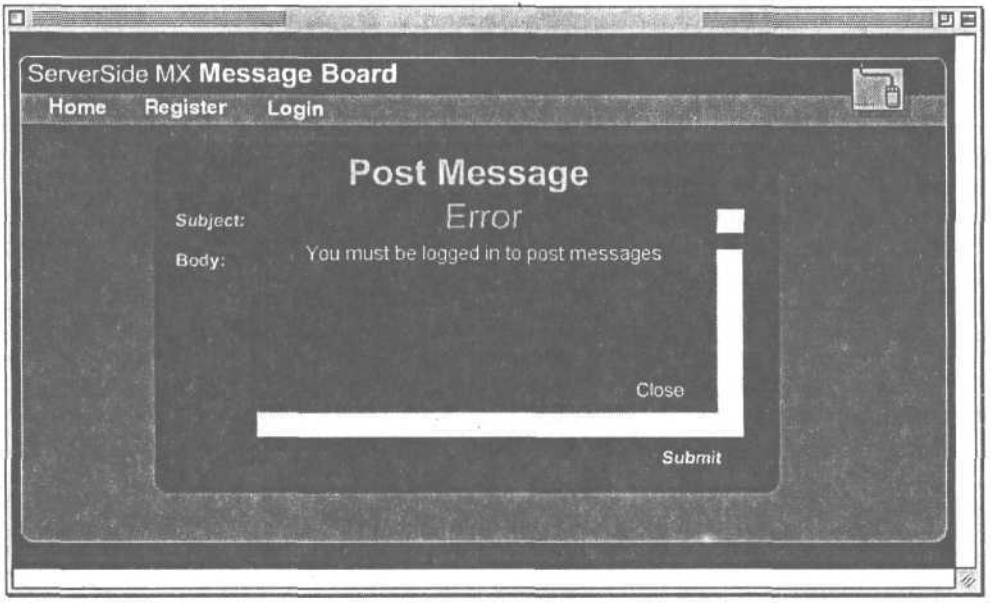

Рис. 3.13. Пользователь, не вошедший в систему, получает сообщение об ошибке при попытке разместить сообщение

Код, приведенный ниже, регистрирует пользователя в базе данных. Процесс регистрации состоит из нескольких выполняющихся последовательно функций. Вначале вызывается функция buildCreateUserTransaction, которой передаются необходимые данные. Эта функция создает XML-документ, необходимый для регистрации. Затем этот документ посылается на сервер. Функция parseRegistrationResults проверяет ответ с сервера, чтобы убедиться в успехе транзакции. После этого программа может перейти к следующему этапу.

```
// Функция registerUser
// Эта функция используется для регистрации пользователя на сервере
function registerUser() {
    // Build the create user transaction .
    var createUserRequest = buildCreateUserTransaction(username.text,
password.text, email.text, showEmail.text);// продолжение строки
    // Зададим документ, который будет послан на сервер
    sd.setDocOut(createUserRequest);
    // Зададим документ, который будет содержать ответ
    sd.setDocIn(new XML());
    // Зададим функцию для события onLoad
    sd.onLoad = parseRegistrationResults;
    // Выполним это
    sd.execute();
} // Конец функции registerUser
// Функция parseRegistrationResults
// Эта функция используется для разбора
// ответа с сервера
function parseRegistrationResults() {
```

```
    // Получим данные от объекта ServerData
    var dataIn = this.getDocIn();
    // Проверим успешность завершения транзакции
    if(!wasSuccessful(dataIn)) {
        // Подключим панель для сообщения об ошибке
        _root.attachMovie("ErrorPanel", "registerErrorPanel",
errorPanelDepth);// продолжение строки
        // Конфигурация панели
        registerErrorPanel.setMessage(errorMessage);
        registerErrorPanel.centerClip();
        return;
    } // конец if(!wasSuccessful(dataIn))
    // Подключим панель для сообщения об успехе
    _root.attachMovie("MessagePanel", "registerMessagePanel",
messagePanelDepth);//продолжение строки
    // Конфигурация панели
    registerMessagePanel.setMessage("You have registered a new account
successfully. Press 'Log In' to continue");// продолжение строки
    registerMessagePanel.setButtonText("Log In");
    registerMessagePanel.centerClip();
    // При закрытии панели перенаправим пользователя
    registerMessagePanel.messagePanelClose = function() {
        gotoAndPlay("Login");
    }
} // Конец функции parseRegistrationResults
// Функция buildCreateUserTransaction
// Эта функция создает XML-документ для нового пользователя
function buildCreateUserTransaction(username, password,
email, showEmail) {// продолжение строки
        // Построим транзакцию 'Create User'
        var createUserRequest = buildBaseRequest("CreateUser");
        // Получим узел данных
        var dataNode = findDataNode(createUserRequest);
        // Создание XML-кода для Username
        // Создадим элемент Username
        var usernameNode = createUserRequest.createElement("Username");
        // Создадим текстовый узел usernameValue
        var usernameValue = createUserRequest.createTextNode(username);
        // Добавим узел usernameValue к узлу usernameNode
        usernameNode.appendChild(usernameValue);
        // Добавим узел usernameNode к узлу данных dataNode
        dataNode.appendChild(usernameNode);
        // Создание XML-кода для Password
        // Создадим элемент Password
        var passwordNode = createUserRequest.createElement("Password");
        // Создадим текстовый узел passwordValue
        var passwordValue = createUserRequest.createTextNode(password);
        // Добавим узел passwordValue к узлу passwordNode
        passwordNode.appendChild(passwordValue);
        // Добавим узел passwordNode к узлу данных dataNode
```

```
        dataNode.appendChild(passwordNode);
        // Создание XML-кода для Email
        // Создадим элемент Email
        var emailNode = createUserRequest.createElement("Email");
        // Создадим текстовый узел emaiWalue
        var emaiWalue = createUserRequest.createTextNode(email);
        // Добавим узел emaiWalue к узлу emailNode
        emailNode.appendChild(emailValue);
        // Добавим узел emailNode к узлу данных dataNode
        dataNode.appendChild(emailNode);
        // Создание XML-кода для ShowEmail
        // Создадим элемент Email
        var showEmailNode = createUserRequest.createElement("ShowEmail");
        // Создадим текстовый узел showEmailValue
        var showEmaiWalue = createUserRequest.createTextNode(showEmail);
        // Добавим узел showEmaiWalue к узлу showEmailNode
        showEmailNode.appendChild(showEmaiWalue);
        // Добавим узел showEmailNode к узлу данных dataNode
        dataNode.appendChild(showEmailNode);
        // Вернем законченный XML-документ
        return createUserRequest;
} // Конец функции buildCreateUserTransaction
```

Заключение

Рассмотренный дискуссионный форум уже готов к размещению, прямо сейчас. Все, что нужно сделать, — это поменять информацию в файле MCProperties.xml, разместить его в соответствующем серверном окружении, и все! Однако, в виде домашнего задания, вы также можете попробовать добавить к программе новые возможности, чтобы сделать ее еще лучше:

- Возможность для пользователя редактировать свои и только свои сообщения.
- Добавить к процессу регистрации возможность выбрать картинку для маркировки сообщений. Вы даже можете дать возможность пользователю загрузить произвольное изображение в формате JPEG.
- Вы можете отображать больше персональной информации о пользователях, например их местонахождение, число размещенных сообщений, несколько электронных адресов и т. д.
- Создать панель администратора, с помощью которой можно ограничить доступ (по паролю) к внутренним функциям форума. Такая панель поможет вам также быстро и легко создавать новые темы, удалять или редактировать сомнительные сообщения, передвигать сообщения между темами (если сообщение не связано с данной темой), а также создать команду администраторов, отвечающих за разные темы.
- Вы можете показывать текущих пользователей в системе и добавить возможность персональных сообщений (внутри самой системы).
- В каждой теме можно поместить интерфейс для интерактивного общения по данной теме, для облегчения общения между пользователями. Протокол (log)

такого интерактивного общения можно рассылать заинтересованным пользователям в рамках рассылки по электронной почте.

Здесь приведены только некоторые идеи. Используйте ваше воображение и положитесь на Flash MX. С такими особенностями, как быстрая обработка XML, прилагаемые компоненты MX, динамическое создание клипов и текстовых полей и форматирование текстовых полей, Flash MX абсолютно подходит для создания интерактивных приложений. Flash MX позволяет создавать интересные программы - пользуйтесь этим и не забывайте про приложенный дискуссионный форум!

4. Аватар-чат [1]

Автор Джоб Макар (Jobe Makar)

В течение вашей Интернет-карьеры вы наверняка заглядывали в какую-нибудь чат-комнату (chat room), иногда называемую просто чатом. Большинство популярных чатов пользуется большой любовью своих посетителей, так как вокруг чата формируется Интернет-сообщество. Такое сообщество - это место, в котором вас всегда ждут и где вы можете внести свой вклад. Они обычно формируются путем сохранения информации о каждом пользователе и возможности для него выразить себя тем или иным способом в рамках некоторого набора параметров (profile), доступного всем остальным для просмотра.

Некоторые экспериментальные чаты дают пользователю возможность выразить себя через персонажей, способных передвигаться в некотором виртуальном пространстве! В качестве персонажей обычно используются животные, инопланетяне или чудовища (monsters), которые выглядят весьма нереально, но тем не менее добавляют новый аспект общения. Вы можете подойти к другому персонажу чата и заговорить с ним! Вы видите людей, заинтересованных в общении с вами, их персонажи находятся рядом с вашим.

В этой главе будет показано, как можно Сделать такой чат, *аватар-чат* (рис. 4.1). У нас будет один персонаж (в качестве примера), и мы не будем собирать какую-либо информацию от пользователей, так как наша программа является только введением в создание аватар-чатов (называемых также чатами с персонажами (character chats).

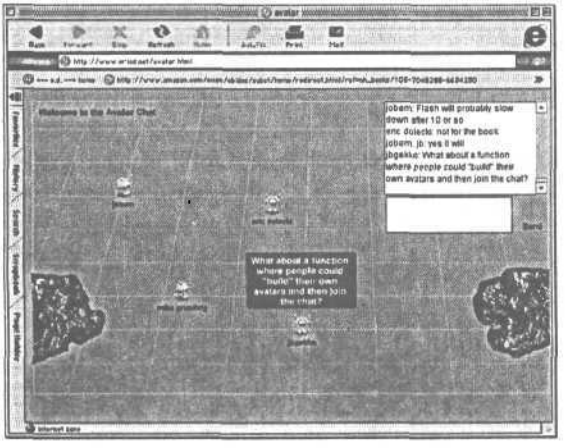

Рис. 4.1. Работающая версия аватар- чата

1. Аватар — виртуальная сущность человека, которая персонифицирует его в сети Интернет. Зачастую представляет собой изображение или анимацию популярного персонажа. Термин происходит из индуизма, где означает воплощение индуистского божества в какое-либо существо. - *Примеч. науч. ред.*

Введение

Перед обсуждением деталей написания программы я хочу перечислить все особенности нашего будущего аватар-чата. Также будут упомянуты источники дополнительной информации, необходимой для полного понимания Данной главы.

Основы чата

Чтобы понять, как создается аватар-чат, сначала нужно понять основы создания обычного чата. Чат, как вы знаете, позволяет сразу нескольким людям обмениваться текстовыми посланиями. Многие чаты также позволяют послать личное сообщение (только одному человеку). Так как же они работают? На ум сразу же приходит мысль о том, что здесь должна присутствовать программа-сервер, соединяющая всех пользователей вместе, - она определяет, кому и когда отправлять сообщение. Это можно реализовать двумя способами:

1. **Использовать в качестве контроллера сценарий на сервере, например ASP страничку.** Это не очень хорошее решение по многим причинам. Такой способ требует обмена большим количеством информации (в отличие от второго способа, описанного ниже). Скорость работы сценария может быть также довольно низкой. Это может подойти для небольшого числа пользователей, но не для реальной чат-системы, которая должна поддерживать сотни пользователей.

2. **Использовать в качестве контроллера сокет-сервер[1] (socket server).** Идея сокета не очень сложна, но ее довольно трудно объяснить. Сокет-сервер является программой (обычно написанной на Java, Visual Basic или C++), которая находится на сервере и следит за определенным портом в ожидании соединений. В среде Flash есть возможность соединения с сокет-сервером по заданному порту. Такое соединение позволяет обмениваться данными с большой скоростью. Сокет-сервер запоминает, кто соединился и когда. Он знает о всех текущих соединениях, именах пользователей (usernames) и их комнатах общения. В приложении под названием "Объект ElectroServerAS" детально описана работа сокет сервера. Для полного понимания данной главы вам необходимо ознакомиться с этим приложением. Клиент (в нашем случае Flash) соединяется с сокет-сервером и пользователю предлагается войти в систему. Для этого может понадобиться набрать имя пользователя и пароль (проверяемых сервером), либо программа просто предложит пользователю некоторое имя на время посещения чата. В большинстве чатов после входа в систему на экране появляется большое окно с текущими сообщениями и текстовое поле для отправки нового сообщения. Обычно вы можете также переходить в другие комнаты (группы общения на разные темы).

1. Сокет - это комбинация IP-адреса и номера порта, которая однозначно определяет отдельный сетевой процесс во всей глобальной сети Интернет. Два сокета, один для хоста-получателя, другой для хоста-отправителя, определяют соединение для протоколов, ориентированных на установление связи. - *Примеч. науч. ред*

Особенности аватар-чата

В нашем аватар-чате только одна комната и все пользователи находятся в ней. После входа в систему перед пользователем появляется новое окно (рис. 4.2), в котором находится его персонаж. Под персонажем указано имя пользователя. Здесь также присутствуют персонажи всех остальных пользователей, подписанные их именами. На правой стороне экрана расположено два текстовых поля. Верхнее поле, которое побольше, является окном чата, отображающим все сообщения. Второе текстовое поле предназначено для набора новых сообщений. Справа от него находится кнопка под названием "send" (отправить). С ее помощью можно отправить набранное сообщение. При получении сообщения оно появляется не только в окне чата, но и над персонажем автора сообщения. Персонажи могут передвигаться по экрану (для этого нужно сделать двойной щелчок мышью в нужном месте).

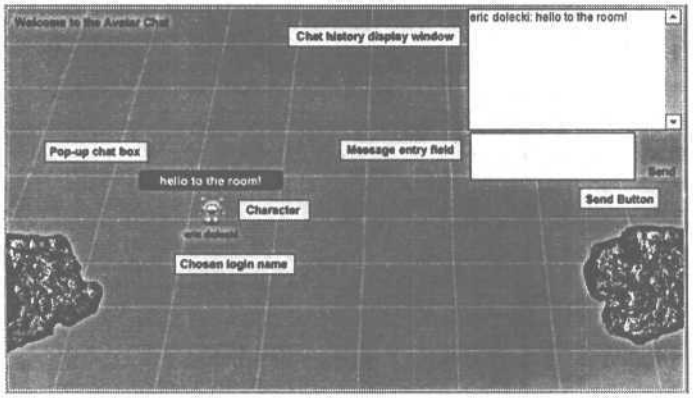

Рис. 4.2. Основные компоненты нашего аватар- чата,

Объект ElectroServer

Нужно упомянуть еще об одной вещи, необходимой нам для работы: объект ElectroServerAS. Сокет сервер, обслуживающий наш чат, называется ElectroServer. Его копия находится на CD-ROM диске. (Детали установки программы ElectroServer описаны в приложении В.)

Клиент Flash обменивается с сокет-сервером пакетами информации в XML-формате. Сервер поддерживает несколько разных типов пакетов, и написание ActionScript-кода для обработки всех типов пакетов может занять довольно много времени. К счастью, вам не нужно этого делать, так как я написал ActionScript объект ElectroServerAS. Этот объект позволяет клиенту легко связываться с сокет-сервером. С его помощью отправка чат-сообщения осуществляется с помощью всего лишь одной строчки ActionScript:

```
ElectroServer.sendMessage("HelloWorld!!");
```

У объекта ElectroServerAS более 40 методов, атрибутов и событий (events). В приложение В описаны все эти методы а также приведены инструкции по использованию

данного объекта. К счастью, объект ElectroServerAS очень прост в использовании и позволяет очень быстро создавать чаты и многопользовательские игры (этот объект также применяется в гл. 5, "Многопользовательская игра").

Приложение В содержит определения действий (actions) объекта ElectroServerAS и описывает, как установить эти действия в окне действий Flash (рис. 4.3), что позволяет не запоминать имена или синтаксис всех действий - можно просто пользоваться окном действий (Actions panel)!

В итоге наш аватар-чат:

- в качестве клиента использует Flash;
- соединяется с сокет-сервером ElectroServer (прилагается на CD-ROM-диске);
- использует объект ElectroServerAS (описан в приложении В);
- позволяет только один персонаж (можно улучшить, добавив персонажи);
- состоит только из одной комнаты (можно добавить любое число комнат).

В последующих разделах данной главы рассматривается архитектура Flash-кода, код ActionScript, отвечающий за перемещение персонажей по экрану, и подробности подключения Flash-клиента к серверу гнезд ElectroServer.

Рис. 4.3. Установка объекта ElectroServerAS позволяет использовать его действия через окно действий Flash (Actions Panel)

Программирование контекста

В этом разделе рассматриваются все аспекты программы, за исключением сетевых (вход в систему, обмен сообщениями и другие взаимодействия между клиентом и сервером, описанные в следующем разделе). Мы рассмотрим разные

компоненты клиента, а также поговорим о персонаже и ActionScript-коде, необходимом для его работы.

Разбор деталей персонажа

Давайте взглянем на структуру клипа персонажа. Найдите файл character.fla (в директории гл. 4) и откройте его. Вы увидите три слоя: Character Actions (действия персонажа), Actions (действия) и Assets (свойства). Слой Character Actions содержит все действия, необходимые для контроля персонажа. В слое Actions находятся все остальные действия, например такое, как инициализация персонажа. В последующих файлах этот слой будет также содержать все действия, непосредственно связанные с чатом. В слое Assets находится клип с именем экземпляра chat.

Внутри клипа chat находится только один слой, содержащий, в свою очередь, другой клип. У второго клипа нет имени экземпляра (instance name), только библиотечное имя (alien) и идентификатор связи (linkage) - alien. Хотя, как вы уже знаете, вызов attachMovie() не требует наличия копии клипа на рабочем поле, это обеспечивает более быструю загрузку в конечном файле. В меню Linkage Properties (свойства связи) Export в первом кадре отключен. Включение экспорта означает необходимость загрузки клипа до загрузки первого кадра, что нам не нужно. В случае отключения экспорта клип загружается по необходимости, когда на рабочем поле появится его копия. Мы пользуемся как раз этой возможностью, так как у нас на рабочем поле есть копия клипа alien. При посещении данного кадра клип alien устанавливается в невидимое состояние.

Сделайте двойной щелчок мышью по клипу alien и посмотрите, что у него внутри. В нем три слоя: Dialog, Character (персонаж) и Shadow (тень). В слое Dialog находится клип с именем экземпляра box. В этом клипе будут отображаться отдельные сообщения чата (объясняется в следующем разделе). В этом слое также есть текстовое поле с именем экземпляра username, в нем будет показано имя пользователя данного персонажа. Слой Character содержит сам персонаж, с именем экземпляра - character. В слое Shadow находится картинка тени.

Сделайте двойной щелчок мышью по клипу character из слоя Character. На монтажной линейке вы увидите 16 помеченных кадров (рис. 4.4). Персонаж отображается под восемью разными углами. В восьми кадрах персонаж стоит (от stand_1 до stand_8), а остальные восемь показывают, как он идет (от walk_1 до walk_8). В каждом кадре есть действие stop() для остановки клипа. Выберите один из кадров, отображающих ходьбу, и сделайте двойной щелчок мышью по находящемуся там клипу. В каждом таком клипе, шесть кадров. Обратите внимание, что у нас только четыре разных клипа передвижения на все восемь направлений, так как каждый клип можно отразить горизонтально и использовать в противоположном направлении. Этот очень простой прием позволяет сократить время написания программы и уменьшить объем файла.

Теперь создайте из этого файла SWF. Вы услышите характерный звук, означающий, что персонаж был создан и появился на экране. Подвигайте мышью, и вы увидите, как персонаж разворачивается, чтобы все время быть лицом к мы-

ши. Двойной щелчок мышью в любом месте экрана приведет к передвижению персонажа в указанное место. Во время перехода на новое место персонаж игнорирует движения мыши и двойные щелчки мышью в других местах (вы не можете на полпути перенаправить его в новое место).

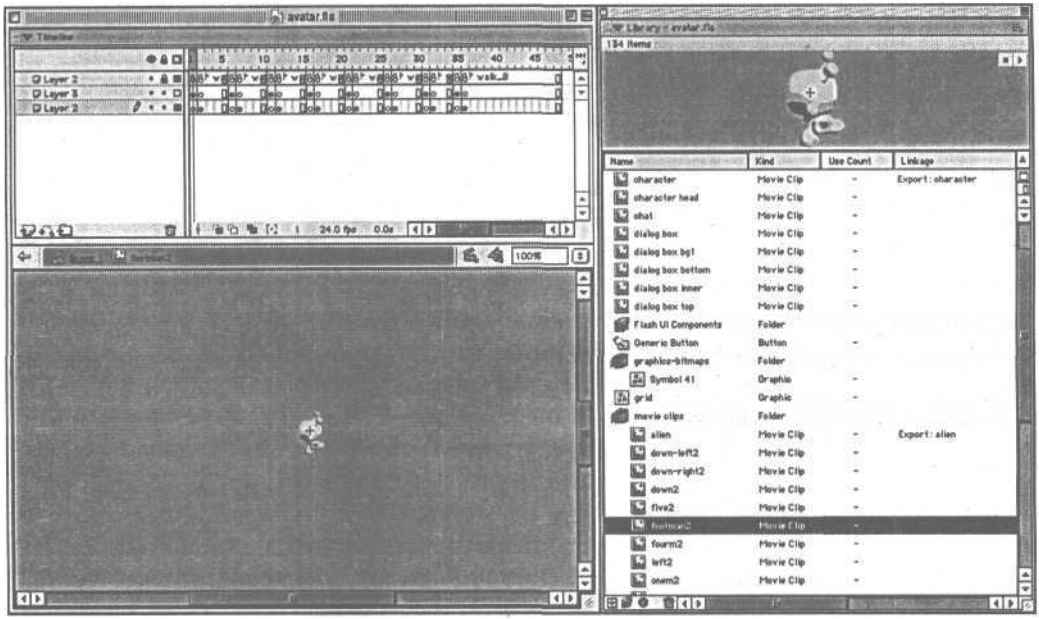

Рис. 4.4. В программе по два кадра на каждый из восьми возможных углов ориентации персонажа: в одном кадре персонаж стоит, а в другом идет

Создание сценариевдействий

Теперь, когда вы увидели персонаж в действии, давайте взглянем на соответствующий ActionScript-код. В течение работы SWF вы видели следующие особенности:

- персонаж был создан,
- под персонажем есть имя пользователя,
- персонаж поворачивается вслед за мышью,
- персонаж переходит на новое место по двойному щелчку мышью.

ActionScript-функция updatePerson() ответственна за создание всех персонажей. Она вызывается сервером ElectroServer, когда в чат-комнате появляется новый человек. Функция updatePerson() также вызывается при двойном щелчке мышью - в этом случае либо существующий персонаж перемещается на новое место, либо (например, в случае нового пользователя) на указанном месте сразу создается новый персонаж. В законченном файле (позднее в этой главе) функция initializeMe() случайным образом выбирает начальное местонахождение персонажа, а затем посылает сообщение о создании нового персонажа. Так как в данном

файле-образце мы не соединены с сервером ElectroServer и не можем получать инструкций с сервера, функция initializeMe() для добавления персонажа вызывает напрямую функцию updatePerson(). Ниже приведена функция initializeMe().

```
function initializeMe() {
    var myCharacter = 1;
    varx = 50+random(500);
    var y = 50+random(300);
    var val = myCharacter+"|"+x+"|"+y+"|0|1";
    updatePerson(name, val);
}
```

Эта функция вызывается при создании нового персонажа. Так как в нашем примере только один персонаж, переменной myCharacter присваивается 1. Если вы решите добавить новые персонажи, вы можете добавить окно, предлагающее пользователю выбрать определенный персонаж. В этом случае в переменной myCharacter будет храниться номер выбранного пользователем персонажа. Этот номер будет передан всем остальным пользователям для отображения в их системах соответствующего персонажа.

Две следующие строчки выбирают случайным образом положение персонажа на экране. Затем в переменной val в виде строки сохраняется следующая информация:

1. Номер персонажа (в нашем случае 1).
2. X-координата.
3. Y-координата.
4. Угол поворота персонажа.
5. Направление взгляда персонажа (от 1 до 8).

В конечном файле эта строка передается на сервер и хранится там как серверная переменная, а в нашем примере она передается функции updatePerson() для создания персонажа с данным именем пользователя в соответствующем месте на экране.

Ниже приведен код функции updatePerson().

```
function updatePerson(name, val) {
var temp = [];
temp = val.split("|");   .
if (people[name] == null) {
        //Создаем персонаж
        people[name] = {};
        if (name == ElectroServer.username) {
            me = people[name];
        }
        people[name].character = temp[0];
        people[name].x = Number(temp[1]);
        people[name].y = Number(temp[2]);
        var angle = Number(temp[3]);
```

```
        var number = Number(temp[4]);
        people[name].username = name;
        //Добавляем персонаж
        chat.attachMovie("alien", name, ++chat.depth);
        people[name].clip = chat[name];
        var clip = people [name].clip;
        initializeCharacter(name);
        clip._x = people[name].x;
        clip._y = people[name].y;
        clip._xscale = 50;
        clip._yscale = 50;
        clip.username.text = name;
        enter = new Sound();
        enter.attachSound("enter");
        enter.start();
    } else {
        //персонаж уже существует, обновляем место на экране
        var endX = Number(temp[1]);
        var endY = Number(temp[2]);
        var angle = Number(temp[3]);
        var number = Number(temp[4]);
        var clip = people[name].clip;
        clip.move(endX, endY, angle, number);
    }}
```

Функция updatePerson() использует объект people для хранения информации о каждом пользователе. Она делится на две части. Если в объекте people еще нет ссылки на данного пользователя (новый пользователь), выполняется первая часть оператора if, в противном случае происходит обновление (во второй части оператора if). Давайте рассмотрим обе части по очереди. Первая часть должна:

- создать в объекте people новую ссылку (на данного пользователя);
- извлечь необходимую информацию из переданной строки val;
- добавить на рабочем поле новый экземпляр клипа character с помощью функции attachMovie();
- Разместить персонаж на экране (масштабировав координаты соответствующим образом);
- Воспроизвести звук, сообщающий о появлении нового персонажа.

Здесь же вызывается функция initializeCharacter(). Она добавляет персонажу события, необходимые для перехвата событий мыши и для контроля поворотов и перемещений. Позже мы обсудим эту функцию более подробно.

Вторая часть функции updatePerson() извлекает информацию из строки val и затем перемещает персонаж путем вызова метода move(), который мы также обсудим немного позже.

Функция initializeCharacter()

Мы рассмотрели ActionScript-код, отвечающий за начальную инициализацию персонажа (добавление клипа и создание объекта). Функция initializeCharacter() обеспечивает персонаж способностью поворачиваться и ходить. А если это собственный персонаж данного пользователя, то он также откликается на двойные щелчки мышью. Ниже приведен код функции initializeCharacter().

```
function initializeCharacter(name) {
        path = people[name];
        path.speed = 5;
        path.locked = false;
        path.angleSpan = 360/8;
        path.clip.ob = path;
        path.clip.move = move;
        if (name == ElectroServer.username) {
        path.myCharacter = true;
        path.clip.onMouseMove = mouseMoved;
        path.clip.onMouseDown = mouseGotClicked;
    }
}
```

Одним из передаваемых параметров является имя пользователя данного персонажа. Создается ссылка (под названием path) на объект с информацией об этом пользователе. Далее задается скорость передвижения персонажа (переменная speed).

Переменной под названием locked присваивается false. Эта переменная употребляется только самим персонажем. Если она равна false, персонаж поворачивается вслед за мышью и отвечает на двойные щелчки мыши, в противном случае персонаж полностью игнорирует мышь. Этой переменной присваивается true на время передвижения персонажа по экрану, чтобы персонаж дошел до конца без странных поворотов по пути (вслед за мышью).

Переменной angleSpan присваивается количество градусов, приходящееся на каждое из восьми направлений. Так как у персонажа восемь кадров поворотов, нам нужно знать, какой кадр поворота показывать. Данная переменная используется именно для этого (об этом будет рассказано немного позже).

На следующей строчке создается ссылка (под названием ob) на объект path в клипе этого же объекта. Это позволяет из клипа персонажа иметь доступ к объекту с информацией о персонаже. Затем в клипе создается функция move(). Она будет вызвана при передвижении персонажа на новое место.

Далее проводится проверка, совпадает ли переданное имя пользователя с именем пользователя данного клиента. В случае совпадения персонажу необходимы дополнительные возможности: повороты вслед за мышью и реакция на двойной щелчок мыши. В этом случае переменной myCharacter присваивается true (означает, что персонаж представляет данного пользователя).

Две следующие строчки ActionScript задают обработчики событий onMouseMove и onMouseDown. Эти обработчики обеспечивают поворот вслед за мышью и переход на новое место.

В результате функция initializeCharacter() задает в клипе character одну функцию (либо три в случае собственного персонажа пользователя).

Функция move()

Давайте взглянем более подробно на некоторые из упомянутых ранее функций. Первая функция, move(), присутствует в каждом персонаже. Ниже приведен ее код.

```
function move(endX, endY, angle, number) {
    //Сохраним переданную информацию
    this.ob.endX = endX;
    this.ob.endY = endY;
    this.ob.angle = angle;
    this.ob.number = number;
    // Рассчитаем скорость передвижения по X и Y
    this.ob.xmov = this.ob.speed*Math.cos(this.ob.angle);
    this.ob.ymov = this.ob.speed*Math.sin(this.ob.angle);
    //Создадим переменные для хранения начального положения
    this.ob.startX = this.ob.x;
    this.ob.startY = this.ob.y;
    //Отправим персонаж на корректный кадр ходьбы
    this.character.gotoAndStop("walk_"+number);
    //Отключим реакцию на мышь
    this.ob.locked = true;
    //Зададим обработчик события onEnterFrame (для передвижения персонажа)
    this.onEnterFrame = function() {
        //Обновим положение персонажа
        this.ob.x += this.ob.xmov;
        this.ob.y += this.ob.ymov;
        this._x = this.ob.x;
        this._y = this.ob.y;
        // Проверим, не дошел ли персонаж до места назначения
        if ((this.ob.endX-this.ob.startX)/Math.abs(this.ob.endX-this.
            ob.startX) != (this.ob.endX-this.ob.x)/Math.abs(this.ob.endX-this.
            ob.x) || (this.ob.endY-this.ob.startY)/Math.abs(this.ob.endY-this.
            ob.startY) != (this.ob.endY-this.ob.y)/Math.abs(this.ob.endY-this.
            ob.y)) {
            this.ob.x = this.ob.endX;
            this.ob.y = this.ob.endY;
            this._x = this.ob.x;
            this._y = this.ob.y;
            this.character.gotoAndStop("stand_"+this.ob.number);
            // Включим реакцию на мышь, если это собственный персонаж
```

```
            if (this.ob.myCharacter) {
                this.ob.locked = false;
                this.mouseMoved();
            }
            this.onEnterFrame = null;
        }
    };
}
```

В данном ActionScript-коде откомментированы все ключевые моменты. При получении с сервера (или из другой функции, как в нашем случае) команды на передвижение вызывается функция move(). Этой функции передаются x- и y-координаты пункта назначения, угол перемещения и направление поворота персонажа (одно из восьми). Сначала эти данные сохраняются в описывающем персонаж объекте (через ссылку od). Далее вычисляются компоненты скорости по x, y и сохраняются как xmov и ymov. Потом сохраняется начальное положение персонажа (для последующего тестирования, не дошел ли еще персонаж до места назначения).

Затем происходит переход на корректный кадр движения персонажа. Это кадр, помеченный "walk_"+number, где number определяет одну из восьми возможных ориентации персонажа. Переменной locked присваивается значение true. Затем задается обработчик события onEnterFrame (для контроля движения персонажа). В первых четырех строках обработчика обновляются координаты персонажа.

Далее идет оператор if, проверяющий, не пора ли остановиться. Этот оператор кажется более сложным, чем на самом деле. Здесь просто проверяется, не изменился ли знак разности между текущей x-координатой и конечной, то же самое для y-координаты. Например, если startX = 10 и endX = 90, их разность (90 - 10 = 80) положительна; если текущая x-координата 92, то соответствующая разность (90 - 92 = -2) отрицательна и персонажу пора остановиться. Содержимое данного if оператора выполняется при достижении цели. Координаты персонажа задаются равными координатам места назначения (endX, endY) и удаляется обработчик события onEnterFrame.

При следующем вызове функции move() снова будет создан обработчик события onEnterFrame и весь процесс повторится снова.

Как видите, это довольно большая функция. К счастью, две следующие значительно меньше. Сейчас мы рассмотрим функцию mouseMoved().

Функция *mouseMoved()*

Функция mouseMoved() вызывается в случае движения мыши (только для собственного персонажа пользователя):

```
function mouseMoved() {
    updateAfterEvent();
    if (!this.ob.locked) {
        var mx = this._parent._xmouse;
        var my = this._parent._ymouse;
```

```
        var xDiff = mx-this._x;
        var yDiff = my-this._y;
        var angle = Math.atan2(yDiff, xDiff);
        var realAngle = angle*180/Math.PI;
        if (realAngle<0) {
            var realAngle = realangle+360;
        }
        var number = Math.ceil(realAngle/this.ob.angleSpan);
        this.character.gotoAndStop("stand_"+number);
        this.ob.angle = angle;
        this.ob.number = number;
    }
}
```

В самом начале проверяется, не блокирован ли персонаж (значение переменной locked; если персонаж блокирован, функция не делает ничего). Далее координаты мыши x и y временно сохраняются в переменных tx и ty. Вычисляя разности между координатами мыши и персонажа, мы можем определить относительный угол линии, соединяющей мышь и персонаж. Этот угол вычисляется с помощью Math-функции, называемой atan2() или arctangent. Она возвращает угол в радианах, так что мы преобразуем его в градусы и присваиваем переменной realAngle. Угол должен лежать в пределах от 0 до 360, но почему-то (как было найдено опытным путем) возвращаемые значения лежат в диапазонах от 0 до 180 и от –180 до 0. Хотя полный диапазон здесь также 360 (от -180 до 180), я предпочитаю работать с углами в диапазоне от 0 до 360, соответствующий оператор if преобразует отрицательные углы в положительные. Затем вычисляется направление (от 1 до 8), определяющее номер кадра анимации персонажа (на основе переменных realAngle и angleSpan, также используя округление). Последние три строчки ActionScript-кода осуществляют поворот персонажа на соответствующий угол и сохраняют значения угла и ориентации.

Функция *mouseGotClicked()*

Это последняя функция в кадре слоя Character Actions (действия персонажа). Она вызывается при нажатии кнопки мыши (в случае собственного персонажа пользователя). Вначале сохраняется время нажатия кнопки. Если с момента предыдущего нажатия кнопки прошло менее половины секунды (500 мс), это означает двойной щелчок мышью и персонажу посылается команда на передвижение. Ниже приведен соответствующий код ActionScript.

```
function mouseGotClicked() {
    if (!this.ob.locked && !_root.chat.offlimits.hitTest(_root._xmouse, _root._ymouse)) {
        this.lastClicked = this.now;
        this.now = getTimer();
        if (this.now-this.lastClicked<500) {
            var mx = this._parent._xmouse;
            var my = this._parent._ymouse;
            if (this.ob.endX != mx && this.ob.endY != my) {
```

```
                this.ob.endX = mx;
                this.ob.endY = my;
                _root.moveMe();
            }
        }
    }
}
```

В самом начале расположен if-оператор, проверяющий две вещи: не блокирован ли персонаж (переменная locked) и не находится ли мышь над местом, запрещенным для передвижения (вызов функции hitTest() объекта клипа). Функция hitTest() возвращает истинное значение если мышь находится над клипом с названием экземпляра offlimits (за пределами). (В нашем файле такого клипа нет, так что функция hitTest() всегда возвращает ложное значение.)

В конечном файле поверх чат-окна будет помещен невидимый клип под названием offlimits, чтобы предотвратить передвижение персонажей в пределах чат-окна. Это очень простой способ ограничения передвижения персонажей.

Затем проверяется разность между текущим временем нажатия кнопки и предыдущим. Значение, меньшее чем половина секунды (500 мс), означает двойной щелчок мышью. Далее координаты мыши сохраняются в переменных mx и ty. Мы сразу же проверяем, не совпадают ли координаты мыши с текущими координатами персонажа, так как в этом случае персонажу не нужно никуда идти. Затем сохраняются координаты пункта назначения (в переменных endX и endY) и вызывается функция moveMe(). Эта функция, в конечном файле, посылает серверу сообщение о передвижении персонажа. В нашем файле-примере это сообщение передается напрямую функции updatePerson().

Мы закончили обзор всего ActionScript-кода, отвечающего за добавление нового персонажа и его передвижение. Давайте теперь рассмотрим ActionScript, связанный с работой всплывающего текстового чат-окна (над персонажем).

Всплывающее текстовое чат окно

При получении чат-сообщения происходит две вещи: сообщение появляется в большом чат-окне с правой стороны экрана (реализовано в конечном файле), и всплывающее текстовое чат-окно (с сообщением) появляется над персонажем автора сообщения. Это всплывающее окно находится на экране около 5 с, а потом исчезает. Высота этого окна задается динамически в зависимости от размера сообщения. В этом разделе мы обсудим детали создания такого окна.

Откройте файл chat_box.fla из директории гл. 4. Этот файл отличается от предыдущего файла-примера только наличием кнопки в нижнем левом углу экрана. Создайте SWF-файл и нажмите на эту кнопку. Над персонажем появится всплывающее окно (рис. 4.5), останется на 5 с и затем исчезнет. Закройте SWF-файл, откройте окно действий (Actions panel) и взгляните на ActionScript-код, соответствующий данной кнопке:

```
on(release) {
chat[name].box.update("Hello, cruel world!") }
```

Этот код просто вызывает метод update() (обновить) экземпляра клипа чат-окна под названием box. Давайте взглянем на этот клип. Откройте библиотеку и найдите клип под названием dialog box. Это тот самый клип, который нам нужен. Сделайте двойной щелчок мышью по этому клипу и ознакомьтесь с его содержимым. В нем два слоя, Actions (действия) и Graphics. Слой Actions, с которым мы вскоре ознакомимся, содержит весь ActionScript-код, необходимый для работы всплывающего окна. Слой Graphics содержит экземпляр клипа также с именем box. Внутри этого клипа три других клипа - top, middle и bottom, представляющие соответственно верх, середину и низ всплывающего чат окна. Три клипа необходимы для того, чтобы можно было масштабировать по необходимости средний клип (в соответствии с размером текста); верхний и нижний клипы обеспечивают закругленные границы.

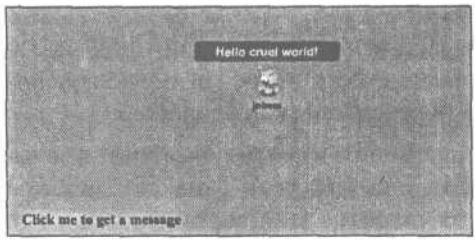

Рис. 4.5. В момент поступления нового сообщения оно также появляется во всплывающем окне над головой персонажа автора сообщения

Теперь давайте взглянем на ActionScript-код слоя Actions. Здесь нет ничего нового или загадочного: масштабирование и размещение клипов, создание текстового поля (с помощью ActionScript) и использование функции setInterval() для вызова функции update() через определенный интервал времени.

Всего здесь четыре функции, которые создают динамически текстовое поле для отображения сообщения, размещают его, задают соответствующий размер фона и устанавливают таймер на удаление окна через 5 с. Ниже приведен весь код.:

```
_visible = false;
function update(newval) {
    if (newval != "" && newval != undefined && newval != null) {
        _visible = true;
        createText(newval);
        startTimer();
    } else {
        clearInterval(displayInterval);
        _visible = false;
    }
}
function startTimer() {
    seconds = 5;
    clearInterval(displayInterval);
    displayInterval = setInterval(update, 1000*seconds);
}
function createText(content) {
    field = "dialog";
```

```
        this.createTextField(field, 1, 10, 0, 290, 0);
        this[field].text = content;
        this[field].multiline = true;
        this[field].wordWrap = true;
        this[field].border = false;
        this[field].autoSize = "left";
        this[field].selectable = false;
        this[field].embedFonts = true;
        this.fieldFormat = new TextFormat();
        this.fieldFormat.color = 0xffffff;
        this.fieldFormat.font = "arial";
        this.fieldFormat.bold = true;
        this.fieldFormat.size = 25;
        this.fieldFormat.align = "center"
        this[field].setTextFormat(this.fieldFormat);
        sizeBG();
    }
    function sizeBGO {
        box.middle._yscale = this.dialog._height;
        box.middle._yscale = this.dialog._height;
        box.bottom._y = this.dialog._height;
        this._y = parent.character._y-30-this._height;
    }
```

Первая строчка (_visible = false) обеспечивает невидимость всплывающего чат-окна в течение загрузки. Мы сделаем его видимым тогда, когда это будет необходимым. Для появления чат-сообщения вызывается функция update() с сообщением в качестве параметра. Если строка сообщения (newval) не пуста, включается видимость чат-окна и вызываются функции createText() и startTimer(). Функция createText() создает текстовое поле и вызывает функцию setBG() (задать фон). Функция setBG() масштабирует средний клип и размещает чат окно на соответствующей высоте. Функция startTimer() с помощью функции setInterval() устанавливает таймер на вызов через 5 с функции update() без параметров. Функция update() воспринимает пустую строку как сигнал отключения видимости чат-окна. Это все! Работа всплывающего чат окна обеспечивается набором очень простых функций.

Как работает чат

Теперь, когда вы знаете, как работает персонаж, пришло время поговорить о деталях работы чата. Создание чата значительно облегчается благодаря объекту ElectroServerAS. (Если вы еще не читали приложение Б, то сейчас самое время его прочитать - либо как минимум будьте готовы использовать это приложение как справочник.) Для тестирования чата вам также необходимо запустить на вашем компьютере сокет-сервер ElectroServer. Как это делается, объяснено в приложении А, "Многопользовательский сервер".

Основные компоненты Flash-файла

Откройте файл avatar.fla из директории гл. 4. Он немного отличается от файлов, которые мы видели раньше. В нем есть предварительный загрузчик нескольких первых кадров. Перейдите к кадру, помеченному done loading, и сделайте двойной щелчок мышью на клипе этого кадра с именем экземпляра chat. В этом клипе находятся все графические элементы. Обратите внимание, что здесь на монтажной линейке четыре помеченных кадра. Давайте рассмотрим их по очереди.

- **failed.** В этом кадре просто находится текст Connection failed. Он показывается, когда не удается установить соединение с сервером.
- **login.** В этом кадре пользователю предлагается набрать имя пользователя и войти в систему.
- **login failed.** Этот кадр отображается после неудачной попытки войти в систему. Здесь есть текстовое поле, в котором выводится причина неудачи. Также здесь есть кнопка с надписью Try Again (попробуй еще раз). При ее нажатии пользователь переходит в кадр login.
- **chat.** Это кадр, в котором пользователь проводит большую часть своего времени. Здесь есть координатная сетка, для того чтобы внести некое ощущение трехмерности в дополнение к трехмерному виду персонажа. На правой стороне экрана находится текстовое окно чата с названием экземпляра window. К этому окну добавлен элемент прокрутки, для просмотра всех сообщений чата. Элементу прокрутки дано имя экземпляра bar, для того чтобы можно было ссылаться на него из ActionScript-кода. Когда приходит новое сообщение и отображается в окне чата, высота элемента прокрутки меняется автоматически и окно прокручивается для показа самого последнего сообщения. В этом кадре также есть другое текстовое поле с названием экземпляра message. В этом поле можно набрать новое сообщение. Когда сообщение готово, его можно отправить нажатием клавиши Enter или кнопки Send. Ниже приведен ActionScript-код, соответствующий кнопке Send.

```
on (release, keyPress "<Enter>") {
    if (message.text != "" && message.text != undefined) {
        _root.chatSend(message.text);
    message.text = "";
    }
}
```

Этот код просто проверяет перед отправкой сообщения, есть ли в текстовом поле какой-нибудь текст. При наличии текста вызывается функция chatSend(), которой передается сообщение. После отправки сообщения содержимое текстового поля удаляется. Мы рассмотрим функцию chatSend() немного позже.

В этом кадре есть также два клипа, которые выглядят как камни. Их названия экземпляров rock1 и rock2. Мы дали им имена, так как они нам будут нужны при сортировке по глубине (Z-sorting). Сортировка по глубине необходима для корректного отображения клипов. Например, если персонаж находится за первым

камнем (rockl), мы вызываем функцию swapDepths() для корректного отображения персонажа за камнем. Этот процесс будет рассмотрен позже.

ActionScript

Перед те, как двигаться дальше, рекомендуется запустить сокет-сервер ElectroServer на порту 1024 (о деталях запуска рассказывается в приложении А) проверить работу чата. Вы можете открыть несколько копий SWF файла для входа в систему сразу нескольких человек. Поэкспериментируйте и исследуйте все возможности чата (рис. 4.6).

Обратите внимание, что в самом верху слоя Actions подключается (#include) файл ElectroServerAS. Этот файл содержит все определения объекта ElectroServerAS и должен быть включен для корректной работы системы.

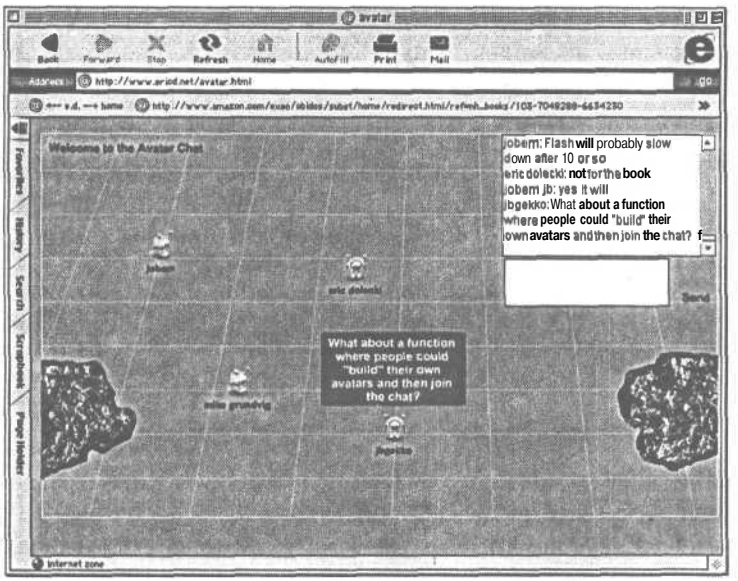

Рис. 4.6. Аватар-чат в действии

С помощью объекта ElectroServerAS можно посылать сообщения, изменять серверные переменные и задавать обработчики таких событий, как прибытие нового сообщения или изменение местонахождения персонажа. Ниже приведен ActionScript-код, задающий обработчики событий и начальные атрибуты объекта ElectroServerAS.

```
ES = new ElectroServerAS();
ES.IP = "localhost";
ES.PORT = 1024;
ES.onConnection = this.connectionResponse;
ES.loginResponse = this.loginResponse;
ES.chatReceiver = this.messageArrived;
ES.onRoomVarChange = this.roomVariablesChanged;
```

```
ES.connectToServer();
```

В первой строчке создается новый объект ElectroServerAS под названием ES. Далее задаются некоторые атрибуты этого объекта: IP-адрес сервера и порт, с которым нужно будет соединиться. Значение порта должно соответствовать порту, на котором сервер гнезд ждет соединений (и который задан в файле атрибутов сокет-сервера). Затем задаются необходимые обработчики событий (подробное объяснение в приложении В). В конце устанавливается соединение с сервером путем вызова метода connectToServer().

Теперь давайте рассмотрим по очереди все четыре обработчика событий. Начнем с функции connectionResponse()/

```
function connectionResponse(success) {
    if (success) {
        chat.gotoAndStop("login");
    } else {
        chat.gotoAndStop("failed");
    }
}
```

Это очень простая функция. В качестве параметра передается true или false. В случае истинного значения соединение успешно установлено и пользователю предлагается войти в систему; в противном случае происходит переход к кадру, помеченному failed.

Функция loginResponse():

```
function loginResponse(success, reason) {
    if (success) {
        chat.gotoAndStop("chat");
    } else {
        chat.gotoAndStop("login failed");
        chat.reason.text = reason;
    }
}
```

Эта функция вызывается, когда с сервера получен ответ на попытку пользователя войти в систему. Первый параметр содержит true или false. Истинное значение означает успешный вход в систему; в этом случае происходит переход непосредственно к окну чата. В случае неудачной попытки входа в систему (success = false) происходит переход к метке "login failed", в этом случае переменная reason содержит описание причины неудачи.

Теперь давайте взглянем на функцию messageArrived()/

```
function messageArrived(info) {
    var from = info.from;
    var body = info.body;
    var msg = from+": "+body+newline;
    chat[from].box.update(body);
```

```
        chat.window.text = ES.addToHistory(msg);
        chat.bar.setScrollPosition(chat.window.maxscroll);
        chatSound = new Sound();
        chatSound.attachSound("chat");
        chatSound.start();
    }
```

Эта функция вызывается в случае получения нового сообщения. Параметр info является ссылкой на объект, содержащий три атрибута: from, type и body. Атрибут from сообщает об авторе сообщения, атрибут body содержит само сообщение. Атрибут type (который мы не используем) содержит тип сообщения: "public" (сообщение для всей комнаты) или "private" (личное сообщение). Данный файл не содержит возможности для отправки личных сообщений (хотя это нетрудно добавить), так что в нашем случае тип сообщения всегда будет "public". Затем создается всплывающее текстовое чат-окно с сообщением над головой соответствующего персонажа, сообщение добавляется в большое чат-окно и историю чата, а положение прокрутки устанавливается на максимум. В конце воспроизводится звук, сообщающий о получении нового сообщения.

Вы, наверно, помните, что сервер гнезд ElectroServer позволяет создавать серверные переменные, связанные с данной комнатой. В случае создания, изменения или удаления серверной переменной об этом извещаются все клиенты комнаты с помощью события onRoomVarChange. В таких переменных хранятся координаты всех персонажей. Это значительно облегчает написание программы. Когда пользователь дает команду своему персонажу передвинуться, изменяется соответствующая серверная переменная и всем рассылается уведомление. Когда пользователь выходит из чата, его переменная автоматически удаляется с сервера и рассылается уведомление всем остальным, так что у них также удаляется соответствующий персонаж. Ниже приведена функция roomVariablesChanged().

```
function roomVariablesChanged(ob, type, varName) {
    if (type == "list") {
        for (var i in ob) {
            updatePerson(i, ob[i]);
        }
    } else if (type == "update") {
        updatePerson(varName, ob[varName]);
    } else if (type == "delete") {
        deletePerson(varName);
    }
}
```

Как уже упоминалось ранее, эта функция вызывается в случае создания, изменения или удаления серверной переменной данной комнаты. При входе пользователя в комнату также вызывается функция VariablesChanged(). Функции roomVariablesChanged() передается три параметра. Первый (ob) является объектом, содержащим все серверные переменные. Следующий параметр (type) является строкой и означает тип события ("list", "update" или "delete"); "list" означает,

что пользователь только что вошел в комнату и требуются все переменные. В этом случае мы проходим в цикле по всем атрибутам объекта и для каждого вызываем функцию updatePerson() с целью создания всех персонажей, "update" означает создание или изменение переменной, и мы вызываем updatePerson(), "delete" означает удаление переменной; и мы должны удалить соответствующий персонаж с помощью функции deletePerson(). Ниже приведена функция deletePerson():

```
function deletePerson(name) {
    chat[name].removeMovieClip();
    delete people[name];
}
```

Это очень простая функция. Она удаляет клип персонажа, а затем удаляет соответствующий объект, который представлял этот клип.

Обратите внимание, что функция moveMe() немного отличается от соответствующей функции в файле character.fla. Тогда из функции moveMe() вызывалась функция updatePerson(), в то время как сейчас вызывается метод createVariable() объекта ElectroServerAS. В результате на сервере создается переменная и появляется событие onRoomVarChange, которое, в свою очередь, вызывает функцию updatePerson(). В конечном итоге вызов moveMe() приводит к вызову функции updatePerson(), но не сразу.

Функция initializeMe() также отличается от функции в файле character.fla. Она добавляет пользователя в данную комнату и создает переменную на сервере.

Теперь давайте взглянем на остальные функции, отвечающие за вход в систему, отправку сообщения и Z-сорт.

```
function login(username) {
    ES.login(username);
}
```

Эта функция вызывается в кадре "login", в качестве параметра передается имя пользователя. Она просто вызывает метод login() объекта ElectroServerAS.

Ниже приведена функция, отправляющая сообщение:

```
function chatSend(info) {
    ES.sendMessage(info, "room");
}
```

Она очень проста. Строка передается в качестве параметра, а затем вызывается метод sendMessage() объекта ElectroServerAS.

Теперь функция, ответственная за Z-сорт (сортировку по глубине):

```
function sortClips() {
    for (var i = 0; i<zSort.length; ++i) {
    zSort[i].swapDepths(zSort[i]. y);
    }
}
```

Каждый раз, когда при помощи функции updatePerson() создается новый клип, в массив zSort добавляется ссылка на новый клип. Функция sortClips() вызывается с частотой два раза в секунду,что задается вызовом функции setInterval() из функции updatePerson(). Z-сорт осуществляется очень простым образом. Экземпляры rockl и rock2 также включены в массив zSort. Z-сорт позволяет добиться более трехмерного вида нашего чат-мира. Персонаж, находящийся логически за другим персонажем, будет отображен соответствующим образом на экране.

Мы рассмотрели весь ActionScript-код данного файла. В этом разделе вы ознакомились с примером практического использования методов и атрибутов объекта ElectroServerAS.

Заключение

В этой главе вы изучили основы создания аватар-чата. К рассмотренному примеру аватар-чата легко можно добавить новые особенности. Вы можете добавить порталы (например, в виде дверей, лестниц и т. д.), в которых пользователь переносится в другую комнату. Комнаты могут отличаться графическими элементами. Можно позволить обмен личными сообщениями (пользователь выделяет другого пользователя с помощью щелчка мыши, а затем посылает сообщение). Желаю удачи!

5. Многопользовательская игра

Автор Михаэль Грюндвиг (Michael Grundvig)

Компьютерные игры всегда были популярны. Я уверен, что при появлении первого CRT-монитора кто-то подумал "На таком мониторе можно сыграть в весьма неплохие игры!". С течением времени сначала видеоигры, а затем компьютерные игры стали играть все большую роль в нашем обществе. Сегодня игровые приставки отличаются от компьютеров в основном только отсутствием клавиатуры и дисковода (floppy). Многие из наиболее популярных игр стали поистине семейными - "Mario Brothers", "Quake" и т. д.

Мало что так повлияло на компьютерные игры, как появление Интернета. Возможность соединяться с другими игроками по всему миру добавила играм много новых особенностей, наиболее выдающаяся из которых - многопользовательская игра. Сегодня многие игры позволяют пользователям играть против других людей через Интернет. Небывалую популярность многопользовательских игр можно приписать только одной вещи: играть с другими людьми очень интересно. Независимо от уровня и возможностей искусственного интеллекта в игре всегда интересней обыграть кого-нибудь из соседей или человека с противоположной стороны света.

Flash также принимает активное участие в этом игровом буме. В Интернете есть много замечательных игр, написанных на Flash. Люди, изучающие Flash, часто в качестве одной из первых программ пытаются написать игру. Хотя однопользовательские игры на Flash довольно широко распространены, многопользовательских Flash игр почему-то очень мало. Данная глава написана с надеждой исправить эту ситуацию.

В этой главе мы напишем многопользовательскую игру под названием "Sea Commander". Эта игра похожа на старую игру Милтона Брэдли "Battleship[1]". Основная идея игры заключается в потоплении вражеских кораблей. У каждого есть игровая доска, разбитая на квадраты (и невидимая для другого игрока), на которой можно размещать корабли. Затем вы по очереди "стреляете" друг в друга, называя координаты вашего выстрела. Если вы попали, ваш противник сообщает вам об этом. Также сообщается о гибели корабля. Игра заканчивается, когда у одного из игроков потоплены все корабли. В нашей игре мы автоматизируем процесс выстрела и нанесения на доску результатов выстрела. Будет использована демонстрационная версия сервера сокетов ElectroServer (основанного на Jave и написанного для использования вместе с Flash) и объект ElectroServerAS.

1. Более известной в России, как Морской Бой. - *Примеч. науч. ред.*

Идеи реализации многопользовательской игры

В пределах среды Flash есть только три реальных способа написания многопользовательской Интернет-игры: опрос (polling), объект XMLSocket или коммуникационный сервер Flash (Flash Communication Server).

Опрос (polling)

В этом случае приложение регулярно посылает серверу запросы на обновление. Обычно это делается с помощью какого-нибудь объекта, поддерживающего загрузку внешних данных, например XML.load() или XML.sendAndLoad(). Клиент делает HTTP-запрос, и сервер приложений возвращает требуемые результаты. Это требует наличия на сервере кода (на ASP, PHP, ColdFusion, Java или на чем, нибудь еще), написанного специально для обслуживания вашей программы.

Недостаток такого метода в том, что запрос на данные исходит со стороны клиента. Клиент не знает, есть ли "свежие данные или нет, он может только спросить. Если провести аналогию, представьте, что вы везете друзей в Лас-Вегас. Ваши друзья спрашивают вас каждые 10 минут: "Мы уже приехали?", а вы всегда отвечаете: "Нет". Когда вы добираетесь до места назначения, вы просто ждете, пока вас не спросят опять, а затем отвечает: "Да". Такой метод очень неэффективен. Он не позволяет серверу уведомить клиента о появлении данных.

В нашей аналогии вы можете приехать в Лас-Вегас сразу после отрицательного ответа на очередной вопрос, но не сможете сказать об этом до следующего вопроса. Метод опроса приводит к задержкам в обмене информацией. Простым решением было бы спрашивать чаще, правильно? Да, это уменьшает задержки, но увеличивает количество выполняемой работы без каких-либо других преимуществ.

Наибольшее преимущество метода опросов в его простоте. Он не требует знания более сложных языков (что требуется в случае XMLSocket). В случае метода опросов не требуется отдельного сервера сокетов. Хватит просто сервера приложений на любых языках.

Коммуникационный сервер Flash

Коммуникационный сервер Flash - это новая технология от фирмы Macromedia, встроенная в последнюю версию Flash player. В сущности это высокопроизводительный сервер сокетов, поддерживающий серверные сценарии и обладающий многими другими особенностями. Так как мы хотим продемонстрировать несколько других технологий, мы не будем использовать коммуникационный сервер Flash для нашей многопользовательской игры, он будет использован позже, в гл. 8.

XMLSocket

Для многопользовательских Flash-игр также можно применять XMLSocket. Он позволяет серверу напрямую уведомить клиента о появлении новых данных. Если вернуться к аналогии путешествия в Лас-Вегас, ваши друзья будут тихо сидеть на

заднем сиденье в течение всего пути. В момент приезда в Лас-Вегас вы сообщаете им о прибытии на место назначения.

Наибольшим недостатком XMLSocket является его сложность. Не то чтобы этот метод особенно сложен, но он сложнее метода опросов. Так как XMLSocket устанавливает сокет-соединение с удаленным сервером, то нужно либо написать, либо взять уже существующий сокет-сервер. Для написания можно использовать многие языки - C++, Java, Visual Basic, C# и т. д., но все они значительно сложнее, чем Flash ActionScript.

Другим недостатком является то, что данный сокет-сервер должен работать постоянно, если вы хотите, чтобы другие люди играли в вашу игру через Интернет. В принципе такие серверы можно запускать на вашем личном компьютере, но для профессионального сайта (production website) этого может оказаться недостаточно. Обычно для этого используется выделенный сервер или заключается контракт с компанией, предоставляющей пользователям машинные/сетевые ресурсы и процессорное время. К сожалению, в обоих случаях требуются деньги.

В этой главе будет использован XMLSocket (так же, как и в предыдущей главе "Аватар-чат"). Мы можем модифицировать код чата из предыдущей главы и добавить поддержку игр. Пользователь сможет бросить вызов другому пользователю (предложить начать игру). К счастью, ElectroServer- и ElectroServer-объект ActionScript (ElectroServerAS) специально настроены под написание многопользовательских игр. Они обладают многими особенностями, облегчающими создание и тестирование сетевых игр. Более подробно эти особенности освещаются в двух приложениях в конце книги.

Введение в игру "Sea commander"

"Sea commander" - это полноценная многопользовательская игра (рис. 5.1). Хотя это явно не следующий "Quake", мы постарались создать достаточно сложный пример. Перед тем как перейти к рассмотрению многопользовательских аспектов игры, необходимо взглянуть на стоящие за ними идеи. Эти идеи относятся не только к данной игре, так что вы сможете без всяких проблем повторно использовать соответствующий код в других проектах.

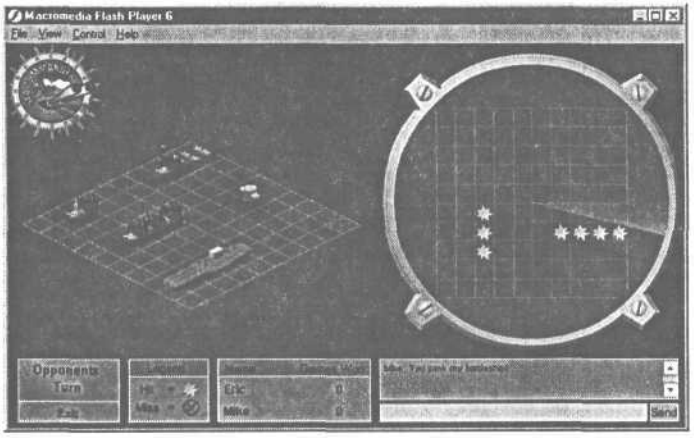

Рис. 5.1. Наша многопользовательская игра в действии

Изометрический мир

Корабли в нашей игре отображаются с помощью изометрической проекции. Благодаря ее простоте изометрическая проекция применяется во многих играх, например таких, как старые версии "Ultima", XCOM, "Civilization 2", XCOM, и многих играх типа RTS ("Real Time Strategy", "Стратегия в реальном времени"). Эту проекцию довольно трудно объяснить словами. Лучше всего это можно представить так: берется квадрат, поворачивается на 90° и сжимается в вертикальном направлении (рис. 5.2).

У изометрических проекций есть одна *очень* удобная особенность: можно ограничиться меньшим количеством нарисованных спрайтов (sprite). Как показано на рис. 5.3, изображения можно очень легко использовать повторно. С помощью одного изображения можно создать и левый и правый вид на судно. Если у нас есть изображение судна сзади, мы также можем применить отражение и получить вид судна спереди. В среде Flash такой метод еще более удобен, так как можно загрузить одно изображение (например, вид слева) и отразить его в коде (для получения вида справа), например таким образом:

```
myObject._xscale -= myObject._xscale * 2;
```

Этод метод применим для многих объектов разного типа и позволяет значительно ускорить время разработки программы.

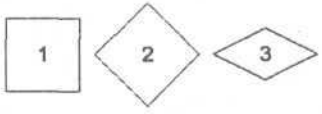

Рис. 5.2. Изометрическую проекцию можно представить как квадрат, повернутый на 90° и немного сжатый в вертикальном направлении

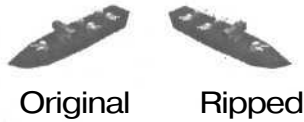

Рис. 5.3. Начальное изображение (вид слева) можно отразить горизонтально и получить вид справа

Размещение объектов в изометрическом мире

Размещение объектов в изометрическом мире не так уж просто. Я не буду углубляться во все подробности, но покажу основной метод размещения. На рис. 5.4 показан игровой экран, на котором размещаются корабли. Перед тем как мы начнем, я хочу поблагодарить Джоба Макара (Jobe Makar) за весь изометрический код, использованный в данном проекте. Похожие программы также написаны и другими людьми, но я обнаружил, что данный изометрический код очень элегантен и удобен для использования. Мне кажется, что это благодаря высшему физическому образованию Джоба.

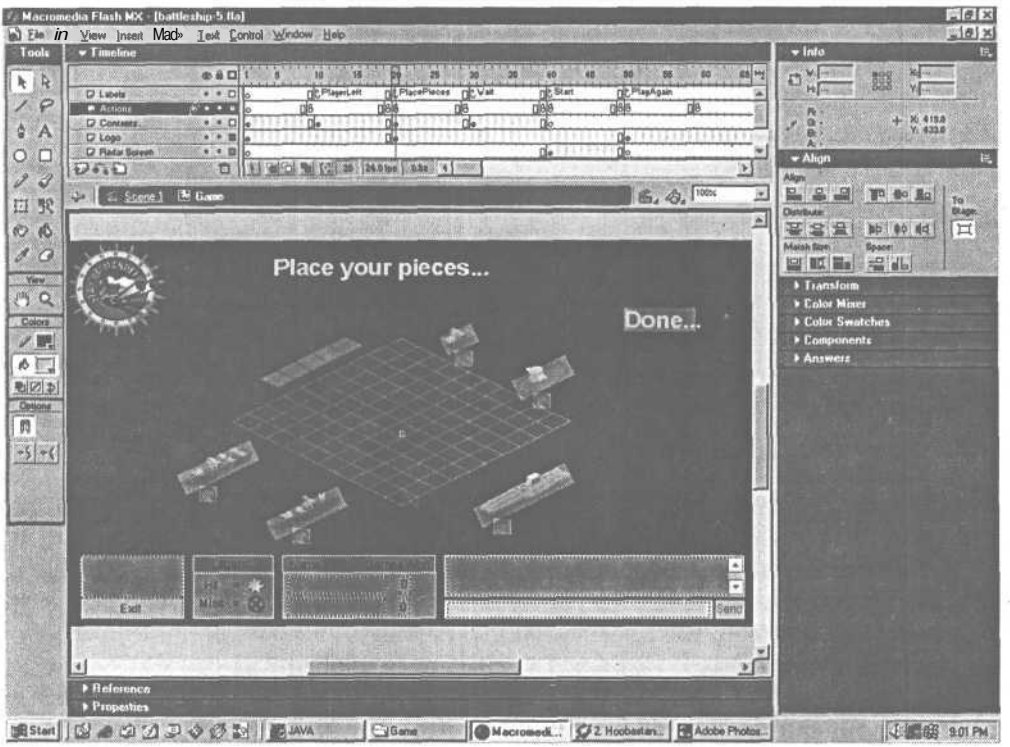

Рисунок 5.4 Игровой экран, на котором размещаются корабли

Изометрическая проекция - это трехмерный мир, отображаемый на двумерном экране. Обычно при программировании в среде Flash вы работаете с двумерной координатной сеткой (для размещения клипов): _x-координата такая-то, _y-координата такая-то и т. д. При работе с изометрической проекцией вам надо в программе иметь дело с виртуальным трехмерным миром. Каждый объект в этом мире имеет три координаты: x, y и z. Также нужна функция для размещения объектов. Если вы будете размещать их сами, они скорее всего будут размещены некорректно. Сначала инициализируем наш изометрический мир

```
yAng = 45;
xAng = 30;
```

```
sinY = cosY=Math.sin(yAng*Math.pi/180);
cosX = Math.cos(xAng*Math.pi/180);
sinX = Math.sin(xAng*Math.pi/180);
```

После инициализации можно пользоваться следующей функцией:

```
function placeObject(name) {
    name._x = (name.z*sinY)+(name.x*cosY);
    name._y = (name.y*cosX)-(((name.z*cosY)-(name.x*sinY))*sinX);
}
```

Как только возникает необходимость в размещении на экране нового объекта, вы просто задаете изометрические координаты данного объекта и вызываете placeObject. Например:

```
root.attachMovie("myObject", "myObject2", 100);
myObject2.x = 0;
myObject2.y = 0;
myObject2.z = 0;
placeObject(myObject2);
```

Обратите внимание, что функции placeObject передается в качестве параметра ссылка на наш объект. Это позволяет корректно размещать объекты из разных монтажных линеек. Объектам в этом случае также проще разместить самих себя:

```
placeObject(this);
```

От экрана к изометрическому миру

Теперь, когда мы рассмотрели демонстрацию клипов на экране с помощью изометрического мира, нам нужно поговорить о том, как сделать обратный переход: взять объект с экрана и перейти к его изометрическим координатам. На первый взгляд может показаться непонятным, зачем это нужно, но на самом деле такая возможность очень полезна. В "Sea Commander" требуется возможность перетаскивания игроком кораблей по экрану для их размещения. При перетаскивании во Flash используются встроенные двумерные координаты _x и _y, но в какой-то момент времени их необходимо перевести в трехмерные координаты объекта x, y и z. Это осуществляется с помощью следующей функции:

```
function screenToIso(fx, fy) {
    zp = (fx/cosY-fy/(sinY*sinX))*(1/(cosY/sinY+sinY/cosY));
    xp = (1/cosY)*(fx-zp*sinY);
    point = newObject();
    point.x = xp;
    pointy = 0;
    point.z = zp;
    return point;
}
```

Здесь есть несколько важных вещей, на которые нужно обратить внимание. Во-первых, возвращается объект point. Это должно быть вам знакомо, так как в Mac-

romedia делается то же самое. Во-вторых, эта функция автоматически предполагает нулевую у-координату. Так как у нас две координаты (fx и fy) преобразуются в три, то нужно задать заранее хотя бы одну координату. В нашем случае мы задаем у-координату ("высота"). При необходимости вы можете варьировать также и эту координату.

Сортировка по глубине

Во многих изометрических программах сортировка объектов по глубине реализуется неправильно. Сортировка по глубине необходима для того, чтобы объекты корректно располагались друг относительно друга. Для решения этой проблемы используется следующая функция:

```
function sortDepth(name) {
    var a = 1000; // максимальная x-координата, константа
    var b = 1000; // максимальная y-координата, константа
    var c = (a * (b-1)) + a; //полная глубина по y
    var n = (c * Math.abs(name.y)) + (a * (Math.abs(name.z)-1)) + name.x;
    name.swapDepths(Math.ceil(n));
}
```

Это довольно уникальный алгоритм. Его основная идея в том, что для всех объектов в изометрическом мире существует общее понятие глубины и на основе (только) самого объекта можно определить его корректную глубину. В результате эта функция работает очень быстро, так как ей не нужно обращаться к другим объектам и выяснять их взаимное расположение с данным объектом.

Основные детали многопользовательских алгоритмов

Мы рассмотрели все предварительные идеи, необходимые для начала игры. Вы уже понимаете детали работы с изометрической проекцией и использования соответствующего кода. Так как в этой главе за основу взят аватар-чат, описанный в предыдущей главе, мы не будем углубляться во все детали. Если вы еще не читали гл. 4, сейчас самое время это сделать.

Так же как и в предыдущей главе, будет использован объект ElectroServerAS на пару с сокет-сервером ElectroServer. Более детально о них можно прочесть в приложениях к данной книге.

В оставшейся части главы мы рассмотрим основные части нашей игры. Для каждой части приведены отдельные фрагменты кода и соответствующие разъяснения.

Место сбора

В каждой многопользовательской игре должно быть место сбора (рис. 5.5), в котором собираются все желающие принять участие в игре. У игроков должна быть возможность найти друг друга и предложить начать игру. Как скажет вам любой игрок *StarCraft* с сервера BNet, в таком месте игроки также хвастаются своими победами или жалуются на поражения.

В предыдущей главе рассказано о создании аватар-чата, но в нашей игре будет использован обычный чат, так как это облегчает переход к игре (набор игроков). Это полноценный и сложный чат, поддерживающий комнаты, списки пользователей, личные сообщения и многое другое. Этот чат демонстрирует возможности объекта ElectroServerAS и является превосходным местом для набора игроков.

Так как в четвертой главе уже были описаны детали соединения с сервером и входа в систему, мы рассмотрим только добавленные изменения.

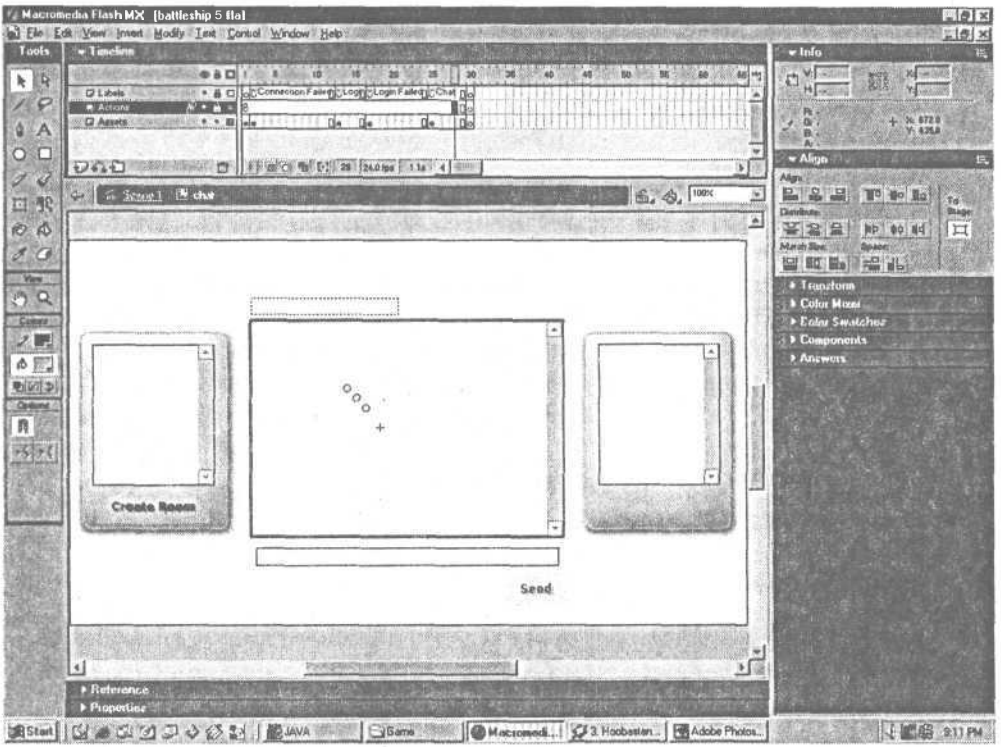

Рис. 5.4. Чат интерфейс в среде Flash

Найдите файл SeaCommander.fla в директории гл. 5 CD-ROM-диска и откройте его. Вы увидите, что в кадрах 1-7 находится предварительный загрузчик кода. Реальный код начинается с кадра 8, где инициализируются все основные чат-функции:

```
// Новый объект ElectroServerAS
ES = new ElectroServerAS();
// Задаем начальные значения
ES.ip = "localhost";
ES.port = 1024;
// Задаем все обработчики событий
ES.onConnection = this.connectionResponse;
ES.loginResponse = this.loginResponse;
ES.chatReceiver = this.messageArrived;
```

```
ES.roomListChanged = this.roomListChanged;
ES.userListChanged = this.userListChanged;
ES.challengeReceived = this.challengeReceived;
ES.challengeAnswered = this.challengeAnswered;
ES.challengeCancelled = this.challengeCancelled;
ES.onConnectionRefused=this.connectionRefused;
// Попытаемся установить соединение
if (!ES.isConnected) {
    // Если мы еще не соединены
    ES.connectToServer();
} else {
    // Если мы уже соединены
    ES.joinRoom("Lobby");
    chat.gotoAndStop("Chat");
    chat.room.text = "Lobby";
}
```

Обратите внимание, что ближе к концу использована ссылка "chat". Это реальный клип чат комнаты. Просмотрите этот клип; в нем находятся все графические элементы и кнопки, необходимые для работы чата. Теперь вернитесь к основной монтажной линейке кадра 8 для рассмотрения использованного кода. Сначала осуществляется попытка соединиться с сервером (с помощью объекта ElectroServerAS, называемого в коде ES). В этом случае возникает событие onConnection. Обработчик данного события приведен ниже.

```
function connectionResponse(success) {
    if (success) {
        chat.gotoAndStop("Login");
    } else {
        chat.gotoAndStop("Connection Failed");
    }
}
```

Здесь происходит переход либо к окну "Login" (в случае успешного соединения), либо к окну "Connection Failed". Предполагая, что соединение произошло успешно, далее вам предлагается войти в систему. В окне "Login" находится кнопка, которая при нажатии вызывает следующую функцию:

```
function login(username) {
    ES.login(username);
}
```

Эта функция посылает серверу запрос на вход пользователя в систему. На сервере для этого есть соответствующее событие под названием loginResponse. Обработчиком этого события является функция с таким же именем:

```
function loginResponse(success, reason) {
    if (success) {
        ES.joinRoom("Lobby");
        chat.gotoAndStop("Chat");
```

```
            chat.room.text = "Lobby";
        } else {
            chat.gotoAndStop("Login Failed");
            chat.reason.text = reason;
        }
    }
```

Так как все пользователи должны иметь разные имена, сервер возвращает истинное или ложное значение, а в случае неудачи также причину неудачи. В случае неудачи осуществляется переход к окну, предлагающему попробовать другое имя пользователя. В случае успеха эта функция переводит вас в комнату под названием "Lobby" и направляет чат клип на метку "Chat".

С этого момента вы можете общаться с другими пользователями. В коде задано также несколько вспомогательных функций для обновления списка комнат, списка пользователей и чат окон. Например, приведенная ниже функция выполняется для обновления списка пользователей.

```
function userListChanged(userList) {
    var path = chat.userList;
    var enabled = path.getEnabled();
    path.setEnabled(true);
    path.removeAll();
    path.setChangeHandler("personClicked", _root);
    for (var i = 0; i<userList.length; ++i) {
        path.addItem(userList[i].name);
    }
    path.setEnabled(enabled);
}
```

Для обновления списка комнат вызывается:

```
    function roomListChanged(roomList) {
        var path = chat.roomList;
        path.removeAll();
        path.setChangeHandler("roomClicked", _root);
        for (var i = 0; i<roomList.length; ++i) {
            var name = roomList[i].name;
            var item = name+"("+roomList[i].total+")";
            path.addItem(item, name);
    }
}
```

Одной из основных является функция для обработки сообщений с сервера:

```
function messageArrived(info) {
    var from = info.from;
    var body = info.body;
    var type = info.type;
    if (type == "public") {
        var msg = formatFrom(from)+": "+formatBody(body)+"<br>";
```

```
} else if (type == "private") {
    var msg = formatFrom(from)+"[private]: "+formatBody(body)+"<br>";
    }
    chat.window.htmlText = ES.addToHistory(msg);
    chat.bar.setScrollPosition(chat.window.maxscroll);
}
```

Здесь сообщение передается в качестве параметра, затем оно форматируется с помощью функции formatFrom, которая просто заворачивает сообщение в HTML-код и возвращает его в виде строки. Далее эта строка отображается на экране. Отправка сообщения также проста:

```
function chatSend(info) {
    ES.sendMessage(info, "room");
}
```

Последней и наиболее важной вещью является реализация отправки вызова на игру и принятие/отклонение вызова от других пользователей.

Для отправки вызова вам нужно только нажать на соответствующее имя в списке пользователей. При этом автоматически вызывается следующая функция:

```
function personClicked(path) {
    var name = path.getValue();
    if (name != ES.username) {
        chat.popup.gotoAndStop("Waiting");
        chat.userList.setEnabled(false);
        ES.challenge(name, "Sea Commander");
    }
}
```

Функция "ES.challenge" допускает любое имя игры. Это сделано специально для поддержки разных игр. В нашем случае мы всегда передаем имя нашей игры "Sea Commander".

Вызов передается другому пользователю и объект ES проверяет, не находится ли он уже в игре. Если данный пользователь уже играет, объект ES автоматически отклоняет вызов. В противном случае вызывается следующая функция:

```
function challengeReceived(from, game) {
    var msg = from+" has just challenged you to a game of "+game+"!";
    chat.userList.setEnabled(false);
    chat.popup.gotoAndStop("Challenged");
    chat.popup.msg.text = msg;
}
```

Как, видите, здесь просто выводится диалог, предлагающий пользователю принять или отклонить вызов. На основе выбора вызывается одна из следующих двух функций:

```
function acceptChallenge() {
    chat.userList.setEnabled(true);
```

```
        chat.popup.gotoAndStop(1);
        ES.acceptChallenge();
        this.gotoAndStop("Game");
    }
    function declineChallenge() {
        chat.userList.setEnabled(true);
        chat.popup.gotoAndStop(1);
        ES.declineChallenge();
    }
```

Эти функции довольно просты. Сначала снова делается доступным список пользователей, удаляется всплывающее окно диалога и посылается ответ о принятии или отклонении вызова. В случае принятия вызова происходит переход к метке "Game". Это место нужно будет изменить в случае добавления новых игр. Вам также нужна будет ссылка на игру, к которой нужно перейти (возможна даже динамическая загрузка соответствующего SWF-файла). При получении ответа на вызов в исходным клиенте выполняется следующая функция:

```
    function challengeAnswered(response) {
        if (response == "accepted") {
            _root.gotoAndStop("Game");
        } else if (response == "declined") {
            chat.popup.gotoAndStop("Declined");
            chat.popup.msg.text = "The challenge has been declined.";
        } else if (response == "autodeclined") {
            chat.popup.gotoAndStop("Declined");
            chat.popup.msg.text = "The challenge has been automatically declined.";
        }
        chat.userList.setEnabled(true);
    }
```

Заметьте, что эта функция также осуществляет переход к метке "Game" в случае принятия вызова. С этого момента оба игрока уже в игре.

Мы рассмотрели основные функции, оставшиеся вспомогательные функции написаны по тому же принципу. Чат также поддерживает создание новых комнат и переход в другие комнаты, послание личных сообщений и другие вещи.

Время на соединение игроков

В этот момент оба игрока готовы к игре. Они приняли вызов и стараются встретиться в соответствующем месте. Все игры проходят в личных комнатах (особенность ElectroServer). Вам, как разработчику, не требуется знать о том, как это реализовано, просто знайте, что игры скрыты от всех остальных пользователей. Это важно, так как на вход в соответствующую комнату требуется время.

Если один игрок соединяется по кабелю, а другой через модем, пользователь с кабелем наверняка будет первым. Поэтому кому-то приходится ждать. Простым решением было бы позволить ожидание в течение заданного периода времени, а за-

тем начать игру, предполагая, что все игроки уже присоединились. Это будет работать, если время ожидания достаточно велико *и* пользователи не потеряли соединения с сервером в результате закрытия окна, отказа браузера или потери соединения с Интернетом.

Это большая проблема, которую довольно трудно решить. Одним из более простых решений (которое использовано в нашей игре) является отслеживание переменных комнаты. Каждый игрок, вошедший в комнату, задает соответствующую переменную. Когда все переменные заданы, можно начинать игру. Более подробно о переменных комнаты рассказывается в приложении, посвященном объекту ElectroServerAS. Ниже приведен соответствующий код.

```
function roomVarChanged(ob) {
    if (ob.player1 == "here" && ob.player2 == "here") {
        locked = false;
        if (!initializedYet) {
            initializedYet = true;
            game.gotoAndPlay("PlacePieces");
        }
    } else {
        locked = true;
        if (initializedYet) {
            game.gotoAndStop("PlayerLeft");
        }
    }
}
function iAmIn() {
    var name = "player"+ES.player;
    var val = "here";
    ES.createVariableф(name, val);
}
```

При входе в комнату выполняется функция iAmIn. Она задает переменную комнаты, что приводит к возникновению события roomVarChanged. Если оба игрока уже в комнате, эта функция вызывает startGame и мы переходим к следующему разделу.

Размещение кораблей

Как только оба игрока вошли в комнату, можно начинать расставлять корабли. Они могут делать это одновременно; все корабли должны быть расставлены до начала игры.

Сначала мы должны подумать о том, в каком виде хранить корабли в памяти. Для этого существует довольно простой метод. Создается двумерный массив с размерами игрового поля. Каждый корабль занимает несколько квадратиков на поле. Так что в методе onClipEvent(load) просто создаем.

```
placementGrid = [[], [], [], [], [], [], [], [], [], [], []];
```

и получаем сетку для размешения кораблей. Когда мы перейдем к коду, размещающему корабли, вы увидите, что данный массив используется очень часто. Теперь у нас есть массив для размещения кораблей, но нам также нужна сетка для отправки нашего игрового поля клиенту противника. Для удобства (причины этого будут ясны позже) позиция корабля в массиве placementGrid будет обозначаться просто ссылкой на сам корабль. Это означает, что вы не сможете послать напрямую массив placementGrid своему противнику, так как в объекте ElectroServerAS используется WDDX ActionScript-реализация от Branden Hall, неспособная передавать клипы. Попытка послать наш массив размещения кораблей приведет к ошибке. Решение очень простое: создаем второй массив:

```
sendToOpponentGrid = [[], [], [], [], [], [], [], [], [], []];
```

В этом массиве положение кораблей отмечается пустыми строками. В результате получается довольно простой объект, который можно послать.

Ранее мы обсуждали код, связанный с изометрической проекцией, а сейчас наступил момент его реального использования. Обратите внимание, что в клипе игры есть клип под названием isoBoard. Если вы взглянете на ActionScript-код этого объекта, вы обнаружите большой оператор for. Этот оператор проходит в цикле по всем пяти кораблям, задает начальные значения соответствующих переменных и создает большую часть функций, необходимых для работы. Каждый корабль накрыт кнопкой со следующим кодом:

```
on(press) {
    startDragging();
}
on(release, releaseOutside) {
    stopDragging();
}
```

Как, видите, нажатие кнопки мыши над кораблем вызывает функцию startDragging, а по окончании перетаскивания вызывается функция stopDragging.

Код функции startDragging приведен ниже:

```
ship.startDragging = function() {
if(this.dragEnabled) {
    this.xOffset = this.p._xmouse-this._x;
    this.yOffset = this.p._ymouse-this._y;
    this.run = this.drag;
    }
};
```

Выглядит довольно просто. В основном потому, что большая часть работы выполняется методом drag. Сама функция startDragging только задает переменные смещения xOffset и yOffset. Эти переменные используются позже для вычисления смещения корабля. Ниже приведен код метода drag.

```
ship.drag = function() {
        this.sortDepth2();
```

```
        this._x = this.p._xmouse - this.xOffset;
        this._y = this.p._ymouse - this.yOffset;
        if (this.useHighlight) {
            this.updateISOCoords();
            this.p.highlight.x = this.round(this.x) * this.p.size;
            this.p.highlight.y = this.round(this.y) * this.p.size;
            this.p.highlight.z = this.round(this.z) * this.p.size;
            this.x = Math.round(this.p.highlight.x / this.p.size);
            this.z = Math.round(this.p.highlight.z /this.p.size);
        if (this.orientation == "vertical" && this.x>=0 &&this.x<=9 &&this.z<=0 && this.z>=-
        10+this.cells) {
            this.p.highlight._visible = true;
            this.p.highlight.showCells(this.cells);
            this.p.highlight._xscale = Math.abs(this.p.highlight._xscale);
            this.p.placeObject(this.p.highlight);
        } else if (this.orientation == "horizontal" && this.x>=0 && this.x<=10-this.cells && this.z<=0 &&
        this.z>=-9) {
            this.p.highlight._visible = true;
            this.p.highlight.showCells(this.cells);
            this.p.highlight._xscale = -Math.abs(this.p.highlight._xscale);
            this.p.placeObject(this.p.highlight);
        } else {
            this.p.highlight._visible = false;
        }
    } else {
        this.updateISOCoordsO;
        this.x = this.round(this.x)*this.p.size;
        this.y = this.round(this.y)*this.p.size;
        this.z = this.round(this.z)*this.p.size;
        this.p.placeObject(this);
    }
};
```

Это довольно большая функция. К счастью, ее код довольно прост. Давайте рассмотрим его более подробно. Вначале выполняется функция sortDepth2 (мы уже рассказывали об этом раньше в разделе, посвященном изометрической проекции). Затем текущее положение корабля задается равным текущему положению мыши. Значение переменной this.p равно значению _parent (здесь родительским объектом является клип isoBoard). В последующих строчках вызывается функция updateISOCoords, которой в качестве параметров передаются новые координаты корабля (в updateISOCoords эти координаты обрабатываются функцией screenToIso, о которой мы уже говорили раньше, а затем они присваиваются переменным this.x, this.y и this.z). Далее выделенный (highlight) клип отображается на экране в соответствующем месте. Большой оператор if используется только для проверки того, что клип находится полностью в пределах игрового поля (в противном случае клип не выделяется).

Мы также упоминали функцию stopDragging:

```
ship.stopDragging = function() {
        if(this.dragEnabled) {
        this.sortDepth2();
            if(this.placed) {
            if (this.placedOrientation == "vertical") {
                for (i=0; i < this.cells; i++) {
                this.p.placementGrid[this.gridX][this.gridY + i] = null;
                }
            } else {
                for (i=0; i < this.cells; i++) {
                    this.p.placementGrid[this.gridX + i][this.gridY] = null;
                }
            }
            this.placed = false;
        }
        this.p.highlight._visible = false;
        this.run = null;
        this.updateISOCoords();
        this.x = this.round(this.x)*this.p.size;
        this.y = this.round(this.y)*this.p.size;
        this.z = this.round(this.z)*this.p.size;
        this.p.placeObject(this);
        this.success = this.p.placeShip(this);
        if (!this.success) {
                this._x = this.startingX;
                this._y = this.startingY;
            this.updateISOCoordsO;
            }
        }
};
```

Еще одна большая функция. Она очень сильно похожа на функцию startDragging, особенно код, связанный с размещением. Давайте рассмотрим ее подробней. Сначала определяется глубина корабля. Это гарантирует правильное расположение корабля относительно всех остальных объектов. Затем нам нужно очистить старое местонахождение корабля. Как только корабль размещен на новом месте, переменной this.placed присваивается истинное значение и нам нужно пройти по всему массиву placementGrid и удалить старые ссылки на корабль.

После удаления (по необходимости) старого размещения корабль размещается на новом месте. Это достигается путем обновления изометрических координат и вызова функции placeObject.

Теперь нам нужно определить, не пересекается ли корабль с каким-нибудь другим кораблем. Для этого вызывается функция placeShip объекта isoBoard. Я не буду здесь обсуждать код данной функции, потому что это просто набор операторов if

и for, которые проходят по доске и проверяют, свободно ли нужное нам место. Если место свободно, эта функция также размещает корабль. Функция placeShip возвращает истинное или ложное значение, в зависимости от того, удалось ли разместить корабль или нет.

Если корабль не удалось разместить (placeShip вернула ложное значение), корабль возвращается на старое место за пределами игрового поля. В противном случае корабль размещен успешно.

После размещения всех кораблей нужно нажать кнопку Done. Эта кнопка вызывает следующую функцию:

```
function lockBoard() {
        // Проверим, все ли корабли размещены
        done = true;
        for(i = 1; i <= 5; i++) {
        if(!this["ship" + i]. placed) {
            done = false;
    }
}
// Удалим кнопки поворота и отключим возможность перетаскивания
if(done) {
    for(i = 1; i <= 5; i++) {
        this["ship" + i].dragEnabled = false;
        this["ship" + i].rotateClip. visible = false;
    }
}
// Возвратим результат
    return done;
}
```

Эта функция довольно проста. Она проходит по всем кораблям и проверяет, что они все размещены. Затем, если все корабли действительно размещены, у каждого корабля удаляются кнопки поворота и отключается возможность перетаскивания.

Если функция lockBoard возвращает true (все корабли размещены), то кнопка затем выполняет следующий код:

```
if(done) {
    myBoardDone = true;
    varobj = new Object();
    obj.action = "PlacePieces";
    obj.placementArray = isoBoard.sendToOpponentGrid;
    sendMove(obj);
    if(theirBoardDone) {
        gotoAndPlay("Start");
    } else {
        gotoAndPlay("Wait");
    }
}
```

Это простой, но довольно важный код. Сначала переменной myBoardDone присваивается true (эта переменная будет использована позже, при получении координат кораблей противника). Далее создается новый объект с переменной action (значение "PlacePieces") и переменной placementArray (со значением из sendToOpponentGrid). После этого вызывается функция sendMove, которая будет рассмотрена в следующем разделе.

Для более полного понимания давайте взглянем на функцию moveReceived. Когда одна сторона посылает свою информацию о размещении кораблей, у другой стороны вызывается функция moveReceived. Ниже приведен фрагмент этой функции:

```
action = obj.action;
if(action == "PlacePieces") {
    theirGrid = obj.placementArray;
    theirBoardDone = true;
if(myBoardDone){
    gotoAndPlay("Start");
}
```

Здесь проверяется значение переменной action (действие). Если это "PlacePieces", то переменной theirGrid присваивается значение переменной placementArray (переданной нам другим игроком). Затем переменной theirBoardDone присваивается истинное значение. Если обе переменные theirBoardDone и myBoardDone истинны, можно начинать игру.

К этому моменту вы должны уже довольно хорошо понимать, как размещаются корабли и каким будет продолжение.

Ход в игре

Реализация хода в игре сложнее, чем может показаться на первый взгляд. При этом нужно учесть много дополнительных деталей.

Основная часть кода игры находится в трех местах: кадр PlacePieces, клип isoBoard и клип игрового поля, находящийся внутри клипа радара. После размещения всех кораблей игрок переходит к кадру "Ready", в котором и проходит собственно сама игра. Соответствующий код:

```
isoBoard._x = -150;
if(myTurn) {
    panel.turn.text = "Your Turn";
} else {
    panel.turn.text = "Opponents Turn";
}
```

Здесь игроку сообщается, чья очередь ходить, а затем клип isoBoard переносится в левую часть экрана. Для определения текущего хода используется переменная myTurn. Она создается в кадре PlacePieces с помощью следующих строк:

```
if(ES.player == 1) {
```

```
        myTurn = true;
    } else {
        myTurn = false;
    }
```

Теперь, когда мы знаем, чей ход, можно перейти к самому ходу (выстрелу по противнику). Сам выстрел делится на две части. Сначала игрок делает выстрел, затем соперник обновляет свое игровое поле в соответствии с выстрелом. Для облегчения обсуждения мы рассмотрим эти две части отдельно.

Выстрел по противнику

Для игрока сделать выстрел очень просто, программа же при этом проходит через несколько разных этапов. При начальной загрузке клипа игрового поля выполняется следующее:

```
currDepth = 100;
g = _parent._parent;
```

Для осуществления выстрела игроку нужно просто нажать мышкой на игровое поле в окне радара. При этом выполняется следующий код:

```
// Обработка выстрела
this.onRelease = function() {
    if(g.myTurn) {
        var xPos = Math.ceil((this._xmouse + 1) / 20) * 1;
        var yPos = Math.ceil((this._ymouse + 1) / 20) - 1;
        // Проверим, не было ли уже выстрелов по этому месту
        var alreadyShot = g.alreadyShot(xPos, yPos);
        if(alreadyShot) {
            g.playSound("AlreadyShot");
            return;
        }
        // Проверим, попал или нет
        var hit = g.takeShot(xPos, yPos);
        if(hit) {
            markShot("hit", xPos, yPos);
        } else {
            markShot("miss", xPos, yPos);
        }
        // Отправим объект и подготовимся к ходу противника
        g.sendMove(obj);
        g.myTurn = false;
        g.panel.turn.text = "Opponents Turn";
    }
}
```

Хотя функция довольно большая, ее код весьма прост. При щелчке мышью на игровом поле в окне радара возникает событие onRelease и выполняется приве-

денный выше код. Вначале проверяется, что сейчас ваш ход. Затем (если ваш ход), вычисляются координаты выстрела, в следующих строчках:

```
var xPos = Math.ceil((this._xmouse + 1 ) / 20) - 1 ;
var yPos = Math.ceil((this._ymouse + 1 ) / 20) - 1 ;
```

После этого проводится проверка, не было ли уже выстрела по данному месту, с помощью функции alreadySpot из кадра PlacePieces:

```
function alreadyShot(x, y) {
    var shot = myShotGrid[x][y];
    if(shot != null) {
        return true;
    } else {
        return false;
    }
}
```

Здесь просто проверяется массив myShotGrid (созданный нами ранее) на предмет сделанного ранее выстрела. Если выстрел уже был, функция возвращает истинное значение и проигрывается звук (выстрел уже сделан); в противном случае выполнение продолжается.

Теперь мы знаем, что на данном месте выстрела еще не было, и можно вызывать функцию takeShot() из кадра PlacePieces. Эта функция, а также вызываемая из нее функция isHit приведены ниже.

```
function takeShot(x, y) {
    var hit = isHitfx, y);
    if(hit) {
        myShotGrid[x][y] = "hit";
    } else {
        myShotGrid [x][y] = "miss";
    }
        return hit;
    }
        function isHit(x, y) {
        var hit = theirGrid[x][y];
        if(hit != null) {
            return true;
        } else {
            return false;
    }
}
```

Функция takeShot в самом начале вызывает isHit для определения, попал или нет. Помните, как в последнем разделе говорилось об отправке координат всех кораблей противнику? Именно эти координаты используются для определения попадания. Это позволяет выполнить данную проверку локально, не обращаясь за информацией к клиенту противника.

Далее на основе информации от функции isHit обновляется массив myShotGrid и возвращается истинное или ложное значение (было попадание или нет).

В коде из клипа игрового поля на основе информации от функции takeShot вызывается функция markShot, чтобы пометить данный квадрат соответствующим образом. Ниже приведен код функции markShot.

```
function markShot(target, gridX, gridY) {
    if(target == "hit") {
        var clip = "HitClip";
    } else {
        var clip = "MissClip";
    }
    currDepth++;
    var newClip = "clip" + currDepth;
    this.attachMovie(clip, newClip, currDepth);
    var c = this[newClip];
    c. x = (gridX * 20) + 10;
    c. y = (gridY * 20) + 10;
}
```

Эта функция берет соответствующий клип (hit/miss, попал или не попал) и присоединяет его к данному квадрату. После этого мы можем, наконец, отправить информацию о ходе нашему противнику:

```
var obj = new Object();
obj.action = "Fire";
obj.x = xPos;
obj.y = yPos;
g.sendMove(obj);
```

Теперь нам осталось подготовиться к ходу противника:

```
g.myTurn = false;
g.panel.turn.text = "Opponents Turn";
```

Сразу после выполнения этого кода ElectroServer отправляет ход другому клиенту и мы переходим ко второй части обработки выстрела, получение выстрела клиентом.

Получение выстрела

Получение выстрела значительно сложней, чем первая часть — нужно проверить, погиб ли корабль или просто получил попадание, включить анимацию, и т. д. При написании игр вы обнаружите, что анимация всегда значительно осложняет самые простые вещи.

В каком месте начинается получение выстрела? Вы угадали, в кадре PlacePieces, а именно в функции moveReceived. Первый раз мы упомянули эту функцию в разделе PlacePieces, так как начальное размещение кораблей на поле в нашей программе одновременно является первым ходом. Ниже приведен соответствующий код:

```
} else if(action == "Fire") {
        myTurn = true;
    panel.turn.text = "Your Turn";
    varx = obj.x;
    var y = obj.y;
    isoBoard.dropBomb(x, y)
```

Сначала обновляется экран (сообщая о том, что наступила ваша очередь ходить). Затем извлекаются координаты выстрела x, y и передаются методу dropBomb клипа isoBoard. При первом вызове функции dropBomb выполняется следующий код:

```
bombDepth++;
this.attachMovie("Bomb", "bomb" + bombDepth, bombDepth);
b = this["bomb" + bombDepth];
b.x = gridX * size;
b.y = -100;
b.z = - gridY * size;
b.rate = 1;
placeObject(b);
```

Это довольно простой код. Здесь на рабочем поле создается клип Bomb в изометрически корректном месте с высотой 100 пикселов над игровым полем. Далее добавляется динамическое событие onEnterFrame со следующим кодом:

```
b.onEnterFrame = function() {
    //Падение закончено
    if(this.y >= -10) {
        // Проверим, попали или нет
        var ship = placementGrid[gridX][gridY];
        // Если попали
        if (ship != null) {
            var explosion = placeExplosion(b.x, b.z);
            ship.hits.push(explosion);
        // Если все части корабля уже подбиты
        if(ship.hits.length >= ship.cells) {
            ship.sunk = true;
            ship._alpha •= 50;
            for(i = 0; i < ship.hits.length; i++) {
                ship.hits[i]. alpha -= 50;
            }
            _parent.sendMessage("You sank my " + ship.name + "!");
            _parent.isGameOver();
        }
    } else {
        placeSplash(b.x, b.z);
    }
    this.unloadMovie();
    }
    this.y += this. rate;
```

```
        this.rate++;
        placeObject(this);
    }
```

Для удобства в коде добавлено несколько комментариев. Основная часть кода посвящена обработке момента окончания падения бомбы. Обратите внимание на следующий фрагмент (после большого оператора if):

```
this.y += this.rate;
this.rate++;
placeObject(this);
```

Этот фрагмент отвечает за ускорение бомбы в процессе падения. Скорость (rate) с каждым кадром увеличивается на 1. При достижении бомбой высоты 10 пикселов начинает выполняться большой оператор if. Внутри этого оператора сначала извлекается ссылка (на данный квадрат) из массива placementGrid. Как вы помните из раздела PlacePieces, в этом массиве содержатся ссылки на корабли либо null. В случае нулевого значения (не попал) нужно только поместить на данное место анимированный всплеск. В противном случае (попал) на это место нужно поместить взрыв.

После размещения взрыва мы добавляем в массив ship.hits данного корабля новое попадание. Если длина этого массива достигла длины корабля (все части корабля подбиты), значит, корабль должен быть затоплен. В этом случае корабль помечается как затонувший, его изображение обесцвечивается (значение _alpha уменьшается на 50), а также обесцвечиваются все клипы попаданий в данный корабль. После этого противнику посылается сообщение о затоплении корабля и вызывается функция isGameOver (рассмотренная в следующем разделе).

Выигрыш

После всего рассмотренного нами выше кода функция, проверяющая, не наступил ли конец игры, кажется очень простой:

```
function isGameOver() {
    var over = true;
    for(var i = 1; i<=5; ++i) {
        var ship = isoBoard["ship"+i];
        if(!ship.sunk) {
            over = false;
        }
    }
    if(over) {
        sendMessage("You jerk! You won!");
        var obj = new Object();
        obj.action = "YouWin";
        sendMove(obj);
        var curScore = parseInt(panel.theirScore.text);
        panel.theirScore.text = curScore + 1 ;
```

```
            playAgainMessage = "You Lost!";
            gotoAndPlay("PlayAgain");
        }
    }
```

Вначале выполняется цикл по всем кораблям, для проверки, все ли корабли потоплены. Если все корабли помечены как затонувшие, то противнику посылается поздравление с победой, наподобие:

```
sendMessage("You jerk! You won!");
```

Это сообщение еще довольно невинно по сравнению с сообщениями, которые могут послать проигравшие игроки. Затем создается объект с переменной action равной "You Win". После отправки этого объекта противнику в его функции moveReceived выполняется следующий код:

```
} else if(action == "YouWin") {
    var curScore = parseInt(panel.myScore.text);
    panel.myScore.text = curScore + 1 ;
    playAgainMessage = "You Won!";
    gotoAndPlay("PlayAgain");
}
```

Здесь обновляется текущий счет и игрок переходит к кадру PlayAgain (начать новую игру). Отсюда при желании можно начать новую игру.

Заключение

Сейчас у вас уже должно возникнуть четкое понимание кода нашей игры, достаточное для создания своих собственных игр в среде Flash на основе сокет, сервера ElectroServer и связанных с ним объектов. Вы видите, сколько много работы требуется для написания игры, и, наверно, понимаете, почему существует не так уж много многопользовательских игр на Flash.

Хотя на это требуется много работы, я думаю конечный результат стоит того. Возможность играть с другими людьми добавляет в игру очень много нового. Без этого вам, скорее всего, очень быстро надоест играть с компьютерными игроками (AI). В случае многопользовательской игры ей обеспечена долгая жизнь и люди будут возвращаться к игре, чтобы попробовать свои силы друг против друга.

Игра, описанная в данной главе, довольно примитивна. К ней можно добавить много новых особенностей и сделать ее гораздо лучше: разные виды оружия, разные способы игры, больше музыки и музыкальных эффектов, расширенный контроль над ходом игры и т. д.

В конце хочу заметить, что, если вам было неинтересно при создании вашей игры, скорее всего другим людям также будет неинтересно в нее играть, так что старайтесь делать все с удовольствием!

6. Мгновенный обмен сообщениями

Автор Михаэль Грюндвиг (Michael Grundvig)

Еще несколько лет назад ваш телефон мог быть занят по нескольку часов сестрой, разговаривающей с друзьями. Сегодня, если у вас дома есть компьютер, то ваша сестра наверняка обменивается сообщениями с помощью программы IM (instant messenger). Вместо нескольких отдельных звонков теперь она может разговаривать со всеми своими друзьями одновременно. Она может общаться с людьми с похожими интересами, при этом никогда не видев их лично. Мгновенный обмен сообщениями начинает играть значительную роль в общении. Он очень популярен, даже фирма Apple добавила соответствующую программу (iChat) в свою операционную систему OSX! Программы мгновенного обмена сообщениями очень популярны также потому, что они очень просты в использовании. Для участия требуется только умение печатать. IM позволяет общаться с людьми из любой точки земного шара.

Когда мы обсуждали, какие именно проекты включить в эту книгу, IM-программа была одной из первых. В среде Flash MX такую программу создавать легче, чем в Flash 5, благодаря дополнительным компонентам, ориентированным на ActionScript и обработку данных. IM, написанный во Flash, можно легко встроить в ваш веб-сайт, позволяя вашим пользователям общаться в реальном времени. Вы можете общаться непосредственно со своими посетителями, без всякой электронной почты. Вы можете сделать свой IM-брэнд и добавить много новых особенностей - гораздо лучше, чем пользоваться чей-то программой с заданным интерфейсом и чужим брэндом (маркой). И вы не сможете встроить чужие IM-программы в ваш сайт или проект - другие IM-программы обычно являются отдельными самостоятельными приложениями.

Наша Flash-программа обмена мгновенными сообщениями

В реализации данной программы применяются технологии Java и XML. Вы увидите позже, как они используются вместе. Это довольно простая IM-программа со многими особенностями, присущими традиционным программам обмена сообщениями:

- созданием записи пользователя (account),
- поддержкой групп,
- возможностью добавлять и удалять друзей,
- списоком друзей,

1. Примером такой программы является всем известная ICQ (она же аська). - *Примеч. науч. ред.*

- иконками текущей активности пользователей,
- обменом сообщениями.

С целью более быстрого написания программы некоторые части интерфейса используют всплывающие HTML-окна (личные чат-окна, окна диалогов и т. д.). Вы можете попробовать собрать все это в единое Flash-приложение с помощью функций loadMovie() или loadMovieNum(). Программа в высшей степени открыта (ориентирована на возможное добавление новых особенностей) - позволяя вам с легкостью вносить свои собственные изменения. Мы также добавили для вас два новых (модифицируемых, skinned) компонента: синий элемент прокрутки и синюю кнопку нажатия (pushbutton). Они находятся во FLA-файле программы.

Краткий обзор IM-программы

При запуске перед вами появляется приветственный экран с тремя кнопками: Login, New User и Help.

Login

При нажатии на кнопку Login появляется всплывающее JavaScript-окно с текстовыми полями ввода для имени пользователя и пароля. Связь с основным окном осуществляется с помощью новой функции localConnection. При нажатии на кнопку Login (уже непосредственно во всплывающем окне) вызывается метод localConnection.send(), который, в свою очередь, вызывает функцию login основного окна. Этот метод передает в основное окно имя пользователя и пароль. В случае ошибки при попытке войти в систему основное окно снова создает всплывающее login-окно и с помощью localConnection.send() задает текст сообщения об ошибке под кнопкой Login.

New User

Нажатие кнопки New User в основном окне приводит к появлению всплывающего JavaScript-окна с интерфейсом для создания нового пользователя. Здесь нет необходимости в localConnection, так как данное всплывающее JavaScript-окно независимо от основного.

Help Button

Эта кнопка ведет к небольшому учебному пособию (tutorial) по программе (перемещение по пособию осуществляется с помощью кнопок Next и Previous).

Теперь давайте перейдем к более интересным аспектам IM-программы.

После входа в систему

После входа в систему перед вами появляется основной интерфейс. В основе этого интерфейса находится список контактов со своим классом под названием ContactList. Все контакты из данного списка можно перетаскивать в разные группы. Сами группы также можно сворачивать и разворачивать. Когда вы видите контакт, с которым хотите поговорить, щелчок мышью по соответствующему имени приводит к появлению окна чата. Здесь используется localConnection (так же, как и в

окне login); однако данный случай немного сложнее, так как одновременно может быть открыто несколько чат-окон (общение с разными людьми). В связи с этим каждому чат окну дается специальный идентификатор, используемый при контактах с основным окном. Новое чат окно в момент своего появления вызывает функцию основного окна, сообщая о себе. Основное окно посылает в ответ уникальный идентификатор, для использования в последующих обменах информацией.

Обновления

С сервера регулярно запрашивается информация о возможных обновлениях, с заданным интервалом между запросами. Мы имеем два вида обновлений. Первым является ChangeState, соответствует изменению состояния контакта в вашем контакт-листе. Вторым является SendMessage, появляется в том случае, когда вам послано сообщение. В случае получения обновления типа SendMessage: если уже открыто чат окно с автором сообщения, то данный автор (контакт) перемещается в самый верх списка контактов и появляется мигающий символ сообщения. При щелчке мышью по контакту исчезает мигающий символ сообщения и появляется чат окно с соответствующим сообщением.

С чего начать

IM-программа обмена сообщениями довольно велика. Она состоит из большого количества кода. Так как кода слишком много, рассмотрены будут только основные части. Мы начнем с идей, заложенных в основу программы, а затем перейдем к самому коду.

Взаимодействие с сервером

Как и большинство программ, описанных в этой книге, IM-программа является клиент-серверным приложением. Вначале осуществляется соединение с сервером и вход в систему. Затем начинается обмен информацией с сервером. Как уже упоминалось ранее, два возможных сообщения с сервера - это ChangeState и SendMessage.

В среде Flash MX существует несколько способов для обмена информацией с сервером. Они описаны в гл. 5, "Многопользовательская игра". В нашей программе используется метод опроса. Вкратце, клиент через определенные интервалы времени посылает на сервер запрос, не появилось ли чего-нибудь нового. Сервер всегда отвечает "да" или "нет". В случае наличия новых данных (ChangeState или SendMessage) эти данные отправляются вместе с ответом сервера. Клиент повторяет этот процесс все время (пока соединен с сервером). После того как клиент соединился с сервером, сервер начинает ожидать запросы от данного клиента (наряду с остальными).

Принимая во внимание большое количество методов во Flash MX, доступных для написания IM программы, неизбежно возникает вопрос, "почему именно метод опросов". Ответ очень прост: метод опросов основан на HTTP и HTTP запросы чаще всего не блокируются программой защиты внутренней сети (firewall). Боль-

шинство других методов обычно основаны на сокетах и поэтому с большой вероятностью могут быть заблокированы.

Обмен данными с сервером

Перед тем как перейти непосредственно к коду программы, давайте рассмотрим, какими именно данными обмениваются клиент и сервер. Далее мы будем называть эти данные транзакциями.

В нашей программе *очень много* транзакций. Для каждого элемента программы существуют свои транзакции, например, для создания новой группы, изменения имени группы, добавления друга, перемещения друга, входа в систему, выхода из системы и т. д. В связи с большим количеством транзакций мы рассмотрим только некоторые из наиболее важных.

Транзакция - это отдельная задача, содержащая запрос на сервер и ответ. И клиент и сервер написаны с расчете на работу с транзакциями. Так как клиент и сервер написаны на разных языках и находятся в разных местах, небходимо выбрать общий формат обмена данными. Для этого идеально подходит XML.

Как уже обсуждалось в предыдущих главах, мы придаем большое значение повторному использованию кода. Для этого желателен общий формат, поэтому мы постарались везде использовать стандартный формат, так что общая структура XML-документов (используемых в транзакциях) вам, наверно, уже знакома.

Login (вход в систему)

Транзакция Login очень проста. Она используется при входе пользователя в систему. Как, видите, в ней содержится только уникальное имя (для идентификации транзакции) и имя пользователя вместе с паролем.

```
<Request>
  <TransactionType>Login</TransactionType>
    <Data>
    <Username>bob</Username>
      <Password>mypass</Password>
    </Data>
  </Request>
```

Сервер отвечает на данный запрос одним из двух документов: Success или Error. В случае успеха сервер возвращает.

```
<Response>
  <Status>Success</Status>
    <Data>
      <Message>The transaction completed successfully</Message>
    </Data>
  </Response>
```

В случае ошибки используется несколько возможных ответов. Если пользователь не существует, возвращается:

```
<Response>
  <Status>Error</Status>
    <Data>
      <Message>Username is not found</Message>
    </Data>
  </Response>
```

В случае неправильного пароля:

```
<Response>
  <Status>Error</Status>
    <Data>
      <Message>Invalid password</Message>
    </Data>
  </Response>
```

CreateUser (создать пользователя)

Транзакция CreateUser применяется для создания нового пользователя. Соответствующий документ содержит имя нового пользователя, пароль и адрес электронной почты:

```
<Request>
  <TransactionType>CreateUser</TransactionType>
    <Data>
      <Username>bob</Username>
      <Password>mypass</Password>
      <Email>bob@aol.com</Email>
    </Data>
  </Request>
```

При получении данного запроса сервер пытается завести нового пользователя. Как и для остальных транзакций, у сервера несколько вариантов ответа: сообщение об успехе или разные сообщения об ошибках - недоступность базы данных, ошибка сервера и т. д.

В случае успеха возвращается:

```
<Response>
  <Status>Success</Status>
  <Data>
    <Message>The transaction completed successfully</Message>
  </Data>
</Response>
```

LoadContactList (загрузить список контактов)

Как только пользователь создал свою учетную запись (account) и вошел в систему, ему нужно в первую очередь получить свой список контактов с помощью транзакции LoadContactList. (В случае нового пользователя сервер возвращает пустой список, так как никаких контактов добавлено еще не было; запрос списка каждым пользователем упрощает код программы.) Сервер хранит список контактов в базе

данных, и по запросу отправляет его клиенту. Клиент посылает на сервер простой документ, типа:

```
<Request>
  <TransactionType>LoadContactList</TransactionType>
  <Data />
  </Request>
```

Сервер в свою очередь отвечает:

```
<Response>
  <Status>Success</Status>
    <Data>
      <Groups>
        <Group ID="19">
          <Name>Authors</Name>
          <Friends>
            <Friend ID="12" State="online">
              <Name>Sean</Name>
            </Friend>
            <Friend ID="34" State="offline">
              <Name>Mike</Name>
            </Friend>
          </Friends>
        </Group>
        <Group ID="134">
          <Name>Budies</Name>
          <Friends>
            <Friend ID="234" State="online">
            <Name>Scott</Name>
          </Friend>
        </Friends>
      </Group>
    </Groups>
  </Data>
</Response>
```

В этом документе содержится много информации. Здесь перечислены все группы из списка контактов пользователя, а также все члены каждой группы. Обратите внимание, что для групп и пользователей также передаются идентификаторы (ID), так как, хотя люди предпочитают иметь дело с именами, код на сервере работает с ID, для обеспечения уникальности записей в базе данных.

SendMessage

После загрузки списка контактов можно начать рассылать сообщения другим людям. Для этого используется транзакция SendMessage, в которой указывается идентификатор (friendID) человека, которому направлено сообщение; friendID в действительности является идентификатором пользователя (userID) в таблице

пользователей в базе данных. Идентификаторы всех друзей были получены с помощью транзакции LoadContactList. Это важно, потому что при получении сообщения в нем содержится friendID отправителя. Ниже приведен XML-документ SendMessage.

```
<Request>
   <TransactionType>SendMessage</TransactionType>
   <Data>
   <FriendID>1</FriendID>
        <Message>Yo</Message>
   </Data>
</Request>
```

При получении этого запроса сервер отвечает стандартным сообщением о том, что все хорошо:

```
<Response>
   <Status>Success</Status>
   <Data>
      <Message>The transaction completed successfully</Message>
   </Data>
</Response>
```

GetUpdates

До сих пор мы рассматривали запросы, отправляемые клиентом, но еще не затрагивали вопроса о том, как именно сервер сообщает клиенту о наличии новых данных (пришло сообщение, изменился статус друга и т. д.). Этот вопрос является основным для IM программы. Сложность в том, что в нашей программе сервер не может сообщить клиенту о новых данных. Так как в основу IM заложен метод опросов, клиент должен сам запрашивать информацию с сервера, поэтому появилась транзакция GetUpdates. В качестве запроса клиент посылает на вид очень простой документ:

```
<Request>
   <TransactionType>GetUpdates</TransactionType>
      <Data />
</Request>
```

И сервер отвечает XML-документом, который может содержать различные новые данные. Этот документ имеет следующую структуру:

```
<Response>
   <Status>Success</Status>
   <Data>
      <Updates></Updates>
   </Data>
</Response>
```

Данный документ отличается тем, что в нем может содержаться несколько разных частей в любом порядке. Это необходимо, потому что один такой документ может содержать сообщения от разных пользователей и изменения в статусе.

Используется два вида обновлений. Одно из них ChangeState, с его помощью сообщается об изменении статуса друзей (online/offline). Выглядит обычно как:

```
<Update>
  <Type>ChangeState</Type>
    <Data>
    <FriendID>10</FriendID>
    <NewState>Offline</NewState>
  </Data>
</Update>
```

Другим является SendMessage. Применяется в том случае, когда кто-то написал вам сообщение. Выглядит обычно как:

```
<Update>
  <Type>SendMessage</Type>
    <Data>
    <FriendID>10</FriendID>
      <Message>hi!</Message>
    </Data>
</Update>
```

Как уже упоминалось ранее, ответ на транзакцию GetUpdates может содержать сразу несколько обновлений. Ниже приведен соответствующий пример.

```
<Response>
  <Status>Success</Status>
    <Data>
    <Updates>
      <Update>
      <Type>ChangeState</Type>
        <Data>
        <FriendID>10</FriendID>
          <NewState>Offline</NewState>
        </Data>
        </Update>
        <Update>
        <Type>SendMessage</Type>
          <Data>
            <FriendID>10</FriendID>
              <Message>hi!</Message>
            </Data>
          </Update>
        </Updates>
    </Data>
</Response>
```

Код программы

Давайте взглянем более детально на то, как клиент соединяется с сервером. Скопируйте с CD-ROM диска директорию Chapters/chapter_6 в вашу рабочую директорию. Давайте откроем в среде Flash MX файл mainWindowWelcome.fla (рис. 6.1) и начнем его разбор!

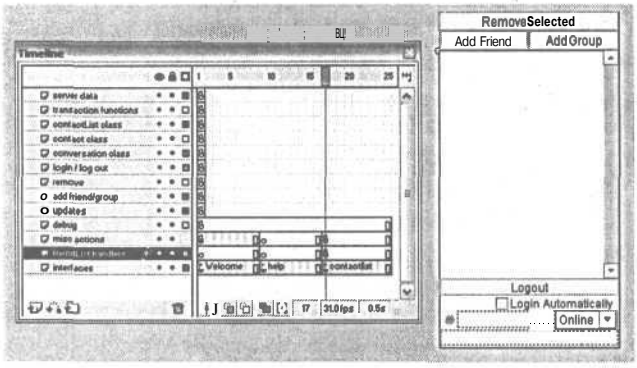

Рис. 6.1 Имена слоев соответствуют их содержанию

Вы сразу видите, что ActionScript-коду посвящено не менее 13 слоев. Вместо размещения в одном кадре по нескольку скриптов, мы разбили все задачи по слоям, чтобы облегчить понимание кода. Имена слоев отражают задачи, выполняемые содержащимся в них кодом.

Server Data

Мы начнем рассмотрение со слоя под названием "server data" (данные сервера). Сдесь содержится код, отвечающий за связь с сервером. Так же как и в других главах, мы используем объект ServerData.as для скрытия деталей реализации. Поэтому в самом начале включается (include) соответствующий файл, а затем выполняется конфигурация объекта:

```
#include"ServerData.as"
_global.server = new ServerData();
server.setMethod(server.SEND_AND_LOAD);
server.setLanguage(server.JAVA);
server.setURL("http://123.123.123.123:8080/flash/tran");
```

Здесь устанавливаются параметры, необходимые для объекта ServerData. Вначале создается новый экземпляр объекта и помещается в глобальную область видимости. Затем задается метод и язык общения. В конце задается URL контроллера транзакций (Transaction Controller) на сервере. Чтобы программа заработала на вашей машине, вам надо будет здесь указать свою информацию (ваш сервер).

Далее в этом слое находится функция под названием sendToServer, которая используется для передачи данных с сервера клиенту. Она упрощает процесс передачи и позволяет избежать дублирования кода. Ниже приведен ее код.

```
_global.sendToServer = function (theXML, callBack,
getUpdates) {
        debug.print(theXML);
        // Проверим тип запроса (на обновление или нет)
        if (getUpdates) endUpdates();
        // Зададим документ для отправки на сервер
        server.setDocOut(theXML);
        // Зададим XML- документ для получения с сервера
        server.setDocIn(new XML());
        // Извлечем аргументы обратного вызова (callback)
        server.callBackArgs = arguments.slice(2);
        // Сохраним функцию обратного вызова
        server.callBack = callBack;
        // Зададим функцию обратного вызова
        server.onLoad = function (success) {
            debug. print(this.getDocIn());
            // Проверим, получен ли какой-нибудь документ
            if (success) {
        // start updates
            startUpdates();
            // call the callback function
            this.callBack.apply(this, this.callBackArgs);
        } else {
        // Вызовем функцию ошибки сервера
        onServerDisconnect();
        }
    }
    // Отправим документ
    server.execute();
}
```

Это довольно большая функция, но она хорошо документирована. Давайте рассмотрим ее подробнее. Эта функция делает довольно много для обеспечения взаимодействия с сервером. В качестве параметров передаются XML-документ для отправки, функция для обработки ответа с сервера и флаг, сообщающий, является ли данный запрос запросом на обновление.

В самой первой строчке XML-документ передается объекту debug. (Мы оставили этот вызов, чтобы показать вам, как отлаживать/трассировать разные виды транзакций.) Следующая строчка выражает довольно сложную идею. Функция endUpdates отключает вызов setTimeout (регулярный вызов функции requestUpdates). Как упоминалось ранее, клиент использует метод опросов и автоматически опрашивает сервер через регулярные интервалы с помощью транзакции getUpdates. Так как транзакция getUpdates также использует функцию sendToServer(), в этом случае нам нужно отключить автоматический вызов функции sendToServer во избежание конфликтов. Эти конфликты, в случае их появления, очень тяжелы для отладки. В качестве примера можно привести случай, когда во время ожидания ответа (на запрос об обновлении) функция requestUpdates снова автоматически вызывается из функции setTimeout. В результате опять вызывается функция sendServerData и ответ на предыдущий запрос будет потерян. Поэтому в случае тран-

закции getUpdates автоматический опрос сервера временно отключается. После получения ответа с сервера автоматический опрос включается снова. Проблема, описанная выше, часто называется в других языках проблемой параллелизма (concurrency).

В следующих двух строчках задаются XML-документы запроса и ответа. Далее извлекаются аргументы обратного вызова (callback) и задается сама функция обратного вызова.

Потом создается новая функция - обработчик события onLoad объекта ServerData. Эта функция используется точно так же, как и в других объектах, поддерживающих onLoad. В начале этой функции стоит очередной отладочный вызов для трассировки данных, полученных с сервера. Далее стоит оператор if, проверяющий, не было ли ошибок низкого уровня при обмене данными с сервером. В случае ошибки вызывается функция onServerDisconnect(), в противном случае снова включаются автоматические запросы на обновление (с помощью функции startUpdates) и выполняется функция обратного вызова, заданная ранее. Как вы видите, это довольно общая функция, подходящая для всех типов транзакций.

Функция onServerDisconnect закрывает соединение с сервером и посылает пользователю сообщение об ошибке. Давайте сначала взглянем на ее код, а потом разберем его более детально.

```
onServerDisconnect = function () {
    var logOutXml = buildLogOutTransaction();
    // Отправим XML-документ
    server.setDocOut(logOutXml);
    server.setDocIn(new XML());
    server.onLoad = null;
    server.execute();
    // Остановим автоматические запросы
    endUpdates();
    // Откроем окно для сообщения об ошибке
    openBrowserWindow("connectionError.html", "connectionError", 147, 91);
    // Выведем ошибку
    doMessage("Server Error");
    // Выйдем из системы
    // Очистим список контактов
    friendsList.removeAll();
    // Удалим сообщение
    doMessage("");
    // Отключим кнопки
    addFriendButton.setEnabled(false);
    addGroupButton.setEnabled(false);
    removeSelectedButton.setEnabled(false);
    // Изменим кнопку Logout на Login
    login.setLabel("Login");
    login.setClickHandler("loginHandler", root);
}
```

Как вы видите, это довольно простая функция. Вначале серверу сообщается о том, что клиент выходит из системы. Важно отметить, что мы не используем для этого функцию sendToServer(). Транзакция Logout посылается напрямую, так как в случае критической проблемы с сервером (отключился от сети или перестал отвечать на запросы) вызов sendToServer приведет к новому вызову onServerDisconnect и мы получим бесконечный цикл. В общем, транзакция Logout посылается напрямую, чтобы не нужно было беспокоиться о ее корректном завершении. После этого с помощью функции endUpdates отключаются автоматические запросы. Далее создается новое окно для сообщения пользователю об ошибке, отключается несколько кнопок, а кнопка Logout изменяется на кнопку Login.

Функции, связанные с транзакциями

Мы рассмотрели несколько наиболее важных транзакций с точки зрения данных, а также рассмотрели процесс отправки транзакций на сервер. Теперь пора обратиться к коду, стоящему за транзакциями.

Все транзакции находятся в слое Transaction Functions, который служит основой всей программы. Мы рассмотрим те же самые транзакции, которые уже были описаны в предыдущем разделе. Как вы знаете, в программе есть также много других видов транзакций. К счастью, они все похожи по структуре, и знание основных транзакций поможет вам разобраться в остальных.

В самом начале слоя включается (include) файл commonTransactionFunctions.as. Вы, наверно, помните этот файл из предыдущих глав. Он включается во все наши программы, так как в нем находится несколько вспомогательных методов для работы с XML. Соответствующая строчка кода:

```
#include "commonTransactionFunctions.as"
```

Вспомогательные методы из данного файла упрощают создание XML-документов. Так как эти методы довольно просты, мы не будем их обсуждать.

Login

Транзакция Login просто передает на сервер имя пользователя и пароль. Соответствующий этой транзакции код создает шаблон документа, добавляет имя пользователя, пароль и возвращает XML-документ:

```
_global.buildLoginTransaction = function (username, password) {
    // Создаем начальный XML-документ
    var createLoginRequest = new buildBaseRequest("Login");
    // Добавляем имя пользователя
    createLoginRequest.addData("Username", username);
    // Добавляем пароль
    createLoginRequest.addData("Password", password);
    // Возвращаем полученный XML-документ
    return createLoginRequest.getDoc(); }
```

CreateUser

Еще одна простая транзакция. Нужно только создать начальный XML-документ, а затем добавить имя пользователя, пароль и электронный адрес. После этого возвращается полученный XML-документ. Соответствующий код приведен ниже:

```
buildCreateUserTransaction = function (username, password, email) {
    // Создаем начальный XML-документ
    var createCreateUserRequest = new buildBaseRequest("CreateUser");
    // Добавляем имя пользователя
    createCreateUserRequest.addData("Username", username);
    // Добавляем пароль
    createCreateUserRequest.addData("Password", password);
    // Добавляем электронный адрес
    createCreateUserRequest.addData("Email", email);
    // Возвращаем полученный XML-документ
    return createCreateUserRequest.getDoc();
}
```

SendMessage

Транзакция SendMessage не сильно отличается от остальных. Обратите внимание, что здесь используется friendID, равный userID (идентификатору пользователя в базе данных). Код, соответствующий этой транзакции:

```
buildSendMessageTransaction = function (friendID, message) {
    // Создаем начальный XML-документ
    var createSendMessageRequest = new buildBaseRequest("SendMessage");
    // Добавляем friendID
    createSendMessageRequest.addData("FriendID", friendID);
    // Добавляем сообщение
    createSendMessageRequest.addData("Message", message);
    // Возвращаем полученный XML документ
    return createSendMessageRequest.getDoc();
}
```

GetUpdates

Транзакция GetUpdates одна из самых простых, так как никаких данных посылать не требуется, только имя транзакции:

```
buildGetUpdatesTransaction = function () {
    // Создаем начальный XML-документ
    var createGetUpdatesRequest = buildBaseRequest("GetUpdates");
    // Возвращаем XML-документ
    return createGetUpdatesRequest;
}
```

LoadContactList

Это тоже простая транзакция. Требуется только задать имя транзакции:

```
buildLoadContactListTransaction = function () {
    // Создаем начальный XML-документ
    var createLoadContactListRequest = buildBaseRequest
    ?("LoadContactList");
    // Возвращаем полученный XML-документ
    return  createLoadContactListRequest;
}
```

Список контактов, контакты и разговоры

Мы рассказали о данных и о том, как они отправляются на сервер. Теперь давайте рассмотрим, как обрабатываются ответы сервера. Для этого создано несколько объектов, - в том числе объекты ContactList (список контактов), Contact (контакт) и Conversation (разговор) при этом имена объектов соответствуют их назначению.

Давайте начнем со списка контактов. Синтаксический разбор списка контактов происходит в функции parseContactList в слое Transaction Functions. Эта функция выполняет "грязную" работу разбора XML-документа и возвращает данные в объекте ContactList. Так как функция довольно длинна, мы постарались добавить достаточное количество комментариев. Более подробное обсуждение последует после кода.

```
parseContactList = function (ContactList, friendsArray, xmlDoc) {
    // Проверим, успешно ли получен документ
    var success = wasSuccessful(xml);
    // если нет, возвращаем false
    if (!success) return false;
    // Получим узел данных документа
    var dataNode = _root.findDataNode(xmlDoc);
    // Получим группы
    var groups = dataNode.firstChild.childNodes;
    // Цикл по группам
    for (var i = 0; i < groups.length; i++) {
        // Получим имя группы
        var groupName = groups[i].firstChild.firstChild;
        // Получим идентификатор группы
        var groupID = groups[i].attributes.id;
        // Добавим идентификатор группы в объект groupID
        groupIDs[groupName] = groupID;
        // Добавим группу в список контактов
        contactList.addGroup(groupName);
        // Добавим в группу друзей
        var friends = groups[i].childNodes[1].childNodes;
        // Цикл по всем друзьям в группе
        for (var j = 0; j < friends.length; j++) {
        // Получим идентификатор друга
        var friendID = friends[j].attributes.ID;
```

```
        // Получим имя друга
        var friendName = friends[j].firstChild.firstChild;
        // Получим состояние друга
        var friendState = friends[j].attributes.State;
        // Создаем новый объект contact
        friendsArray.push(new contact(friendName, friendID,
        ?friendState));
        // Задаем обработчик события onMessageReceived
        friendsArray[friendsArray.length - 1].onMessageReceived
        ?= function (contact) {
           friendsList.moveContactToTop(contact.group, contact);
           friendsList.update();
        }
        // Добавляем контакт в группу
        contactList.addContact(groupName, friendsArray
        ?[friendsArray.length - 1]);
        }
    }
    // Возвращаем true (успешное завершение)
    return true;
}
```

В качестве параметров передается ссылка на список контактов (contactList), массив всех друзей, известных системе, и XML-документ для обработки. Если XML-документ содержит ошибку с сервера, то функция сразу же возвращает false, для соответствующей обработки ошибки в вызывающей функции.

Далее начинается реальная обработка XML-документа. Сначала извлекается информация о группах. Как вы помните, все друзья разбиты по группам, а список контактов состоит из групп. Данный код извлекает информацию о группах и добавляет в объект contactList с помощью метода addGroup. Позже мы обсудим более подробно класс contactList.

Затем идет цикл по всем друзьям в данной группе. Как и в случае групп, извлекается идентификатор и имя, вдобавок к этому извлекается статус друга. На основе этих данных создается новый объект Contact, а потом добавляется в массив friendsArray, переданный в виде параметра. Затем к текущему объекту Contact добавляется обработчик события onMessageReceived. Этот обработчик вызывается в том случае, когда приходит сообщение от данного контакта. Обработчик очень простой - он просто передвигает контакт в начало списка друзей группы и обновляет этот список. Следующая строчка после обработчика вызывает метод addContact объекта ContactList, чтобы добавить данный контакт в список контактов.

Теперь, когда мы рассказали о том, как обрабатываются XML-данные, давайте рассмотрим сами объекты, начиная с класса ContactList.

ContactList

Это довольно большой класс. Вместе с комментариями он насчитывает почти 600 строк. Из этих 600 строк более 250 приходятся на метод update(), который создает клипы на экране. Так как он довольно велик и содержит большое количество комментариев, мы не будем обсуждать его детально. Вместо этого давайте рассмотрим основные методы и как они используются.

addGroup

Метод addGroup добавляет в список контактов новую группу. Ниже приведен его код.

```
contactList.prototype.addGroup = function (groupName) {
    // Добавим группу в массив групп
    this.groups.push({name: groupName, contacts: [], open: false});
    // Обновление
    this.update();
    // Вызываем событие onChange
    this.onChange();
    // Вызываем событие onAddGroup
    this.onAddGroup(groupName);
}
```

Этот метод довольно прост. В качестве параметра передается имя группы (как строка). Вначале создается новый объект и добавляется в массив групп. Важно отметить, что у нас нет отдельного объекта для групп. Для простоты воспользуемся сокращениями, предоставляемыми Flash MX, для создания нового объекта снесколькимипеременными:

name. - просто имя группы;

contacts. - массив объектов Contact внутри группы;

open. - булево значение, указывающее, открыта ли группа на экране пользователя.

После добавления группы в массив вызывается метод update() для обновления списка контактов на экране. В конце вызывается обработчик события onAddGroup.

addContact

Этот метод добавляет контакт в группу и обновляет список контактов на экране. Ниже приведен его код.

```
contactList.prototype.addContact = function (group, contact, moving) {
    // Сохраняем в контакте группу
    contact.group = group;
    // Проверим, передана ли группа в качестве параметра, если нет,
        // то добавим в первую группу
        if (group == null) {
            groupName = this.groups[0].name;
        } else {
            groupName = group;
```

```
        }
        // Найдем группу в массиве
        for (var i in this.groups) {
            // Проверим группу
            if (this.groups[i].name == groupName) {
            // Добавляем контакт
            this.groups[i].contacts.push(contact);
            // Выходим из цикла
            break;
        }
    }
    // Вызываем событие onChange
    this.onChange();
    // Если контакт не передвигается, вызываем событие onAddContact
    if (!moving) this.onAddContact(group, contact);
}
```

В качестве параметров передается группа (group), объект contact и булево значение (переменная moving), сообщающая, нужно ли просто передвинуть контакт в группе. Вначале создается ссылка на массив групп, затем выясняется имя группы. Если переданная переменная group равна null, то контакт добавляется в первую группу массива, - это происходит при добавлении в первый раз друга в список контактов.

Установив имя группы, мы проходим в цикле по всем группам в поиске данной группы. Как только группа найдена, добавляем контакт в массив contacts (из найденной группы) и выходим из цикла.

Далее вызывается событие onChange объекта ContactList. В конце, если контакт добавляется (а не просто передвигается), вызываем функцию onAddContact, передавая в качестве параметров группу и контакт.

Свернуть и развернуть группу

Графический интерфейс нашей IM-программы поддерживает сворачивание и разворачивание групп на экране. Как вы уже видели, во всех объектах group есть переменная под названием open.

Для сворачивания и разворачивания групп используется несколько методов, в зависимости от ситуации. Все они работают по одному и тому же принципу, поэтому мы рассмотрим только самый сложный:

```
contactList.prototype.toggleExpandCollapse = function (group) {
    // Цикл по всем группам (поиск группы)
    for (var i = 0; i < this.groups.length; i++) {
        // Проверка имени группы
        if (this.groups[i].name == group) {
        // Проверяем, развернута ли группа
        if (this.groups[i].open) {
            // Изменяем на противоположное значение
```

```
            this.groups[i].open = false;
        } else {
            // Изменяем на противоположное значение
            this.groups[i].open = true;
        }
            // Выходим из цикла
            break;
        }
    }
    // Обновляем группу и контакты
    this.update();
    this.onChange();
}
```

Методу toggleExpandCollapse в качестве параметра передается имя группы. Вначале выполняется поиск этой группы (в цикле по всем группам). Затем проверяется, развернута ли группа на экране (open = true) или свернута (open = false). Далее состояние группы меняется на противоположное и происходит выход из цикла.

В конце вызываются методы update() и onChange(). Метод update() обновляет состояние групп на экране, проходя в цикле по всем группам и перерисовывая их на основе переменной open.

Для изменения состояния группы на экране также можно использовать методы expandGroup() (развернуть) и collapseGroup() (свернуть). Если вам уже известно, что группа развернута (или свернута), то можно вызвать один из этих методов.

update

Этот метод является самым большим во всей нашей программе. Тем не менее он довольно прост для понимания благодаря большому количеству комментариев в FLA-файле. Для краткости мы будем приводить фрагменты кода без этих комментариев. За фрагментами следует их обсуждение.

Метод update обновляет список контактов на экране. Для большей элегантности мы используем некоторые новые особенности среды Flash MX.

Метод начинается с цикла по всем группам, для каждой группы выполняется следующий код:

```
var target = this.container.createEmptyMovieClip("group_" +
?this.groups[i].name, ++num);
target.createTextField("groupName", 1, 12, 0, 110, 20);
target.groupName.selectable = false;
target.groupName.embedFonts = true;
var groupFormatting = new TextFormat();
groupFormatting.color = 0xff0000;
groupFormatting.font = "Arial";
target.groupName.text = this.groups[i].name;
target.groupName.setTextFormat(groupFormatting);
target.attachMovie("openStatus", "openStatus", 2);
```

```
target.openStatus._y = 5;
if (this.groups[i].open) {
    target.openStatus.gotoAndStop(1);
} else {
    target.openStatus.gotoAndStop(2);
}
target._y = Math.round(target._height * num);
target.thisRef = this;
target.group = this.groups[i].name;
```

Как видите, вначале создается новый клип и присваивается переменной под названием target. К этому клипу добавляется текстовое поле под названием groupName, затем создается объект textFormat и применяется к только что созданному текстовому полю.

Далее клип присоединяется к группе (для иллюстрации, развернута группа или свернута). Если группа развернута, клип переходит ко второму кадру.

В конце задается положение клипа в списке контактов и клипу предоставляется доступ к некоторым значениям списка контактов.

Следующий фрагмент кода добавляет в текущую группу обработчик события onRelease, позволяющий изменить состояние группы с помощью щелчка мыши.

```
target.onRelease = function () {
this.thisRef.toggleExpandCollapse(this.group);
this.thisRef.selected = this.group;
this.thisRef.update();
}
```

Данная функция просто вызывает toggleExpandCollapse и передает соответствующую группу в качестве параметра. Обратите внимание, что ссылка thisRef указывает на объект ContactList. В конце вызывается метод update для обновления списка контактов на экране (чтобы отразить новое состояние группы).

Приведенный ниже фрагмент кода задает цвет фона текстового поля groupName.

```
if (this.selected == this.groups[i].name) {
    var tempFormat = new TextFormat();
    tempFormat.color=0xFF9900;
    target.groupName.setTextFormat(tempFormat);
}
```

Далее в методе update идут следующие строчки:

```
if (this.groups[i].open) {
for (var j = 0; j<this.groups[i].contacts.length; j++) {
```

Здесь проверяется, развернута ли данная группа, и если да, то следует цикл по всем контактам в группе. Такая проверка необходима, так как показать контакты нужно только для открытых групп.

Следующий фрагмент кода создает графический вид контакта. Он очень похож на приведенный ранее код, относящийся к группам, так как решает похожую задачу.

```
target = this.container.createEmptyMovieClip("group_" + this.groups[i].name + "_" + this.groups[i].contacts[j].name, ++num);
target.createTextField("contactName", 1, 12, 0, 110, 20);
target.contactName.selectable = false;
target.contactName.embedFonts = true;
contactFormatting = new TextFormat();
contactFormatting.font = "Arial";
target.contactName.text = this.groups[i].contacts[j].name;
target.contactName.setTextFormat(contactFormatting);
target.attachMovie("status", "status", 2);
target.status._y = 5;
target.status.gotoAndStop(this.groups[i].contacts[j].contactStatus);
target._y = Math.round(target._height * num);
target._x = 15;
target.thisRef = this;
target.contactRef = this.groups[i].contacts[j];
target.groupRef = this.groups[i];
```

Сначала создается новый клип, затем к нему добавляется текстовое поле. Это текстовое поле форматируется с помощью объекта TextFormat. Затем к клипу присоединяется новый клип под названием status. Клип status показывает, находится ли данный человек в сети (online). В конце клип размещается на экране и к нему добавляется несколько ссылок на необходимые объекты.

Следующий фрагмент кода также похож на код, относящийся к выделению группы. Здесь проверяется, выделен ли контакт, и (если выделен) изменяется цвет фона:

```
if (this.selected == this.groups[i].contacts[j]) {
    var tempFormat = new TextFormat();
    tempFormat.color = 0xFF9900;
target.contactName.setTextFormat(tempFormat);
}
```

В методе update есть также два обработчика событий для данного контакта - onPress и onRelease. Мы не будем приводить их код, так как он написан по тому же принципу, по которому написан и рассмотренный ранее код.

Обработчик события onPress вызывается при нажатии кнопки мыши над контактом. Он используется для перетаскивания контактов из одной группы в другую. Этот обработчик создает событие onEnterFrame, которое используется для проверки, не находится ли контакт над какой-нибудь группой, и (если да) выделяет соответствующую группу.

Обработчик события onRelease вызывается при окончании перетаскивания. В нем удаляется событие onEnterFrame, заданное в событии onPress, а затем, при необходимости, контакт переносится в другую группу.

На этом мы закончим рассматривать метод update. Объект ContactList также составляет значительную часть нашей программы. Нам осталось рассмотреть буквально пару объектов.

Класс Contact

Этот класс в основном используется для хранения данных. В нем есть только один метод. Код данного класса находится в слое Contact Class.

Сам класс хранит имя пользователя, идентификатор (ГО) и статус. Ниже приведен код метода messageReceived:

```
contact.prototype.messageReceived = function (message) {
// Проверим, разговаривает ли с кем-нибудь данный контакт
if (this.conversation != null) {
    // Направим сообщение в текущий разговор
    this.conversation.messageReceived(message)
} else{
    // Зададим статус
    this.setStatus("message");
    // Сохраним сообщение
    this.message = message;
}
this.onMessageReceived(this);
}
```

Этот метод вызывается при получении нового сообщения. Основная идея в том, что полученное сообщение является частью разговора и данный Contact-объект сам управляет своими текущими разговорами. Наша реализация IM-программы поддерживает только разговор один на один. Это означает, что Contact-объект (контакт) может ссылаться максимум на один Conversation-объект (разговор). Если эта ссылка равна null, то статус контакта задается равным "message". Это приведет к появлению мигающей иконки контакта (пришло новое сообщение). Если Contact объект уже содержит ссылку на какой-то текущий разговор (Conversation-объект), то сообщение передается методу messageReceived данного Conversation- объекта.

Conversation-класс

Все управление текущим разговором осуществляется в объекте Conversation. Этот объект служит интерфейсом между окном списка контактов и окном текущего разговора. Создание нового объекта Conversation автоматически приводит к появлению нового окна разговора.

В этом объекте хранятся сообщения (в виде очереди) от человека, с которым ведется разговор. Это необходимо по двум причинам: возможно получение нескольких сообщений еще до начала разговора (и все их нужно сохранить) и окно разговора может быть закрыто в любое время (игнорируя возможные новые сообщения).

Одним из наиболее важных методов класса Conversation является messageReceived(). Этот метод вызывается объектом Contact при получении сообщения. Ниже приведен код метода.

```
conversation.prototype.messageReceived = function (message)
{
    if (this.connected) {
        var contactName = String(this.contact.name);
        var contactID - String(this.contact.id);
        var contactStatus = String(this.contact.contactStatus);
        this.connection.send(this.id + "main", "messageReceived", {name: contactName, id: contactID,
        contactStatus: contactStatus}, message);
        this.checkMessageId = setInterval(this, "checkMessage", 100, message);
    } else {
        this.queue.push(message);
    }
}
```

В самом начале оператор if используется для определения, соединен ли объект с окном разговора. В случае наличия соединения (через localConnection) Conversation-объект извлекает необходимые данные из Contact-объекта и форматирует сообщение для отправки в открытое окно с помощью ссылки на объект LocalConnection (под названием this.connection) и метода send этого объекта. В конце устанавливается таймер для вызова через 100 мс метода checkMessage, код которого приведен ниже.

```
this.checkMessage = function (message) {
if (!this.connection.connected)
    this.onMessageFailure(this.contact, message);
    clearInterval(this.checkMessageId);
}
```

Это необходимо для того, чтобы открытое окно получило сообщение. Если метод checkMessage обнаруживает ошибку (потерю соединения), то вызывается событие onMessageFailure. Последняя строчка удаляет таймер, так как повторные вызовы не нужны.

Возвращаясь к методу messageReceived, нам нужно еще обсудить вторую половину оператора if:

```
this.queue.push(message);
```

Эта строчка добавляет сообщение в массив queue, чтобы не потерять сообщение при отсутствии окна разговора. При следующем открытии окна в нем будут отражены все сообщения, накопившиеся в массиве queue.

Отображение всех сообщений из массива queue выполняется в следующем коде:

```
conversation.prototype.doQueue = function () {
    // Цикл по массиву queue
    for (i = 0; i < this.queue.length; i++) {
```

```
            // Получим данное сообщение
            this.messageReceived(this.queue[i]);
        }
    }
```

Здесь для каждого сообщения из массива queue вызывается метод messageReceived.

Заключение

IM программа является довольно сложным приложением. В этой главе мы рассмотрели основные вещи, необходимые для написания клиента. Мы не стали обсуждать детали реализации на стороне сервера, так как это необязательно и слишком усложнило бы главу. Весь код серверной части программы полностью документирован и находится на приложенном CD-ROM-диске. При условии знания соответствующего языка у вас не должно быть никаких проблем в понимании этого кода.

С помощью экспериментирования и некоторого исследования вы можете значительно расширить возможности программы. Вы можете связать ее с Flash Communication Server и добавить видео чат, связать с Flash Remoting или сделать что-нибудь еще. В любом случае мы надеемся, что описанная в данной главе программа-пример поможет вам!

7. Клиент электронной почты

Автор Тим К. Чанг (Tim K. Chung)

Почти у каждого посетителя Интернета либо есть электронный адрес Hotmail, либо он знает человека с таким адресом. Программы электронной почты типа Hotmail и Yahoo предлагают возможность посылать и получать сообщения, находясь в сети, на основе централизованного места хранения данных. Популярность таких программ очевидна - люди со всего мира могут обмениваться письмами просто заходя на центральный сервер.

В этой главе описана архитектура простой программы электронной почты и средства ее создания на основе Flash MX, XML, Java и Microsoft Access. Программа работает с вашей электронной почтой на основе уже существующего электронного адреса.

Сначала мы взглянем на программу в общем, а затем рассмотрим транзакции данных и как все работает. Последний раздел главы посвящен некоторым методам программирования в среде Flash, применяемых в описываемой программе (в том числе прототипы и локальные соединения), а также классам, созданным при работе над программой.

Хотя описываемые технологии относятся к среде Flash MX, вы также познакомитесь с архитектурой почтового клиента и сможете сделать то же самое на основе других технологий (ColdFusion, .NET, mySQL или каких-нибудь еще).

Основы работы Peachmail

Для начала давайте рассмотрим все шаги, выполняемые пользователем при работе с программой электронной почты типа Hotmail, основанной на браузере. Мы назвали нашу программу Peachmail. Познакомьтесь с Джо, новым пользователем нашей программы.

Мотивация Джо

На дворе лето! Джо собирается в отпуск и должен оставить свой компьютер дома. Он осознает, что может потерять доступ к своей электронной почте, и пытается найти решение. Друг говорит ему о Peachmail, новой услуге, позволяющей получить доступ к своей электронной почте из любого места (с помошью браузера и веб-сайта Peachmail).

Джо решает проверить.

Регистрация Джо

Джо возвращается домой, включает компьютер и заходит на веб-сайт Peachmail. Вначале ему нужно зарегистрироваться, предоставив свое имя пользователя, электронный адрес, пароль, POP-сервер и SMTP-сервер. В качестве имени пользователя Джо выбирает имя mejoe. Теперь, когда у него есть регистрационная запись на сервере, Джо входит на Peachmail-сервер под своим новым именем.

Вознаграждение

Теперь у Джо появился доступ к следующим услугам Peachmail:

- Адреснаой книге. Джо может добавлять, редактировать и удалять адреса из своей адресной книги.
- Электроннаой почте. Джо может просматривать, получать, отправлять, размещать по различным папкам и удалять электронные сообщения.
- Модификациям регистрационной записи. Джо может изменить свою персональную информацию или даже удалить свою запись.

Джо добавляет в адресную книгу адреса нескольких друзей и посылает сообщение о том, что у него останется доступ к электронной почте:

```
Hey you crazy blokes! It's me Joe! I'm using Peachmail.com
to access my email. It's absolutely brilliant!
Cheers,
Joe
```

После этого Джо выходит с Peachmail-сервера и решает отметить свой успех в кофейне Starbucks.

Более детальный взгляд: то, что неизвестно Джо

В процессе освоения Peachmail Джо на самом деле имел дело с тремя разными интерфейсами:

- Клиентом, или GUI (графический интерфейс пользователя). Это то, что появилось перед Джо после загрузки веб-сайта Peachmail. Клиент форматирует окно программы, взаимодействует с Джо и посылает команды Джо на сервер. Клиент может быть написан на чем угодно, начиная с HTML-форм и заканчивая Flash-программой или Java-аплетом.
- Серверной частью. Сервер выполняет действия на основе команд клиента, управляет данными, пересылаемыми между клиентом и местом их хранения, и регулирует обмен информацией с клиентом. Серверная часть обычно написана на языках сценариев, таких, как Perl, ASP, PHP или ColdFusion, хотя она также может быть написана на основе более общих языков, например Java или C++.
- Интерфейсом хранения данных, таким как текстовый файл или база данных. Здесь хранятся все данные. Самыми распространенными базами данных

являются Oracle, MSSQL Server для больших проектов и mySQL, postgreSQL или Microsoft Access для средних и малых проектов.

В нашей программе в качестве клиента служит Flash MX, на сервере применяется Java, а базой данных является Microsoft Access. Как вы увидите позже, обмен данными между клиентом и сервером происходит в XML-формате в соответствии с методами обработки XML-данных в среде Flash MX. Теперь давайте посмотрим, чего мы хотим от нашей программы.

Основные требования

Подведем краткий итог действий Джо при работе с Peachmail: он зарегистрировался, вошел под своим именем пользователя и отправил сообщение. На основе этого можно составить блок-схему действий программы (рис. 7.1).

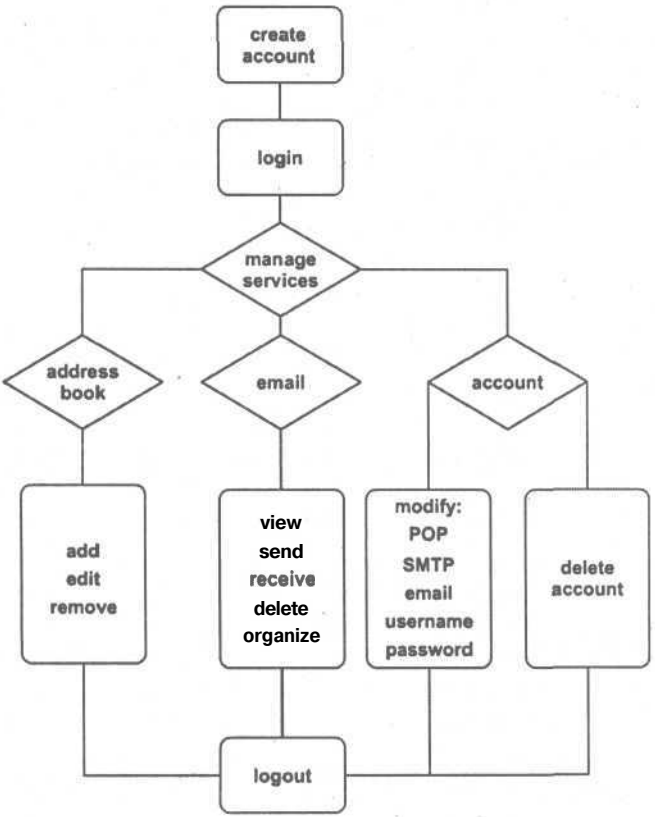

Рис. 7.1. Блок-схема работы почтового клиента

На рисунке приведена работа идеального почтового клиента. В реальной программе для простоты введены следующие ограничения:

- Peachmail не поддерживает модификацию регистрационной записи.

- Адресная книга в Peachmail не поддерживает редактирование.
- В Peachmail почта хранится только в одной папке; не поддерживается создание новых папок и перемещение сообщений.
- Сохраняются только приходящие сообщения.
- Письма с вложениями можно только получать, но не отправлять.

После обсуждения базы данных будет рассмотрен процесс регистрации и входа под своим именем, а также услуги адресной книги и почты. В качестве упражнения вам предоставляется возможность улучшить Peachmail и превратить программу в идеального почтового клиента (как описано в блок-схеме).

Теперь, получив хорошее представление о поведении программы, мы можем сосредоточиться на данных, необходимых для ее работы.

Определение данных

Первым пунктом в схеме стоит регистрация. Так как у нас почтовый клиент, вдобавок к непременным имени пользователя и паролю Джо нужно также указать свой электронный адрес, SMTP-сервер и POP-сервер. Вся эта информация, а также уникальный идентификатор пользователя (UserID) будут храниться в одной таблице под названием Users. Поля этой таблицы показаны на рис. 7.2.

Рис. 7.2. Таблица Users

Когда Джо входит под своим именем пользователя, программа проверяет по таблице Users, что Джо является зарегистрированным пользователем. Любые изменения, сделанные Джо в своей регистрационной записи, сохраняются в этой таблице.

После входа под своим именем Джо получает доступ к своей адресной книге, являющейся просто списком имен и электронных адресов знакомых Джо. Для хранения имен и адресов мы создаем таблицу под названием AddressBookEntries, добавляя идентификатор записи AddressBookEntryID. Также добавляем поле UserID для ссылки на соответствующего пользователя, чтобы знать, кому принадлежит данный адрес. Поля этой таблицы приведены на рис. 7.3.

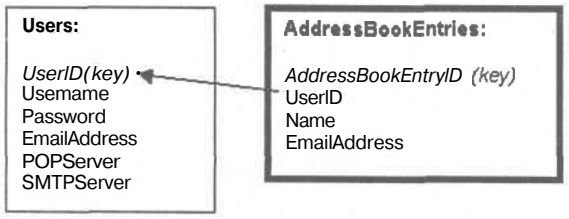

Рис. 7.3. Таблица AddressBookEntries

В основе Peachmail лежат почтовые услуги, используемые Джо для получения и отправки почты. Полученные сообщения сохраняются в таблице под названием Messages. Поля этой таблицы показаны на рис. 7.4.

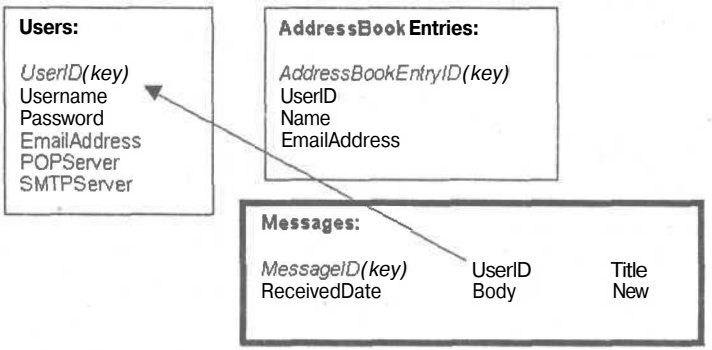

Рис. 7.4. Таблица Messages

В поле Body хранится само сообщение, в поле ReceivedDate - дата получения. Заголовок находится в поле Title, а поле New отмечает еще не прочитанные сообщения. Поле UserID указывает на регистрационную запись автора сообщения, а в поле MessageID хранится идентификатор сообщения.

В базе данных также есть простая таблица под названием Folders, позволяющая Джо создавать новые папки для хранения сообщений. Эта таблица показана на рис. 7.5.

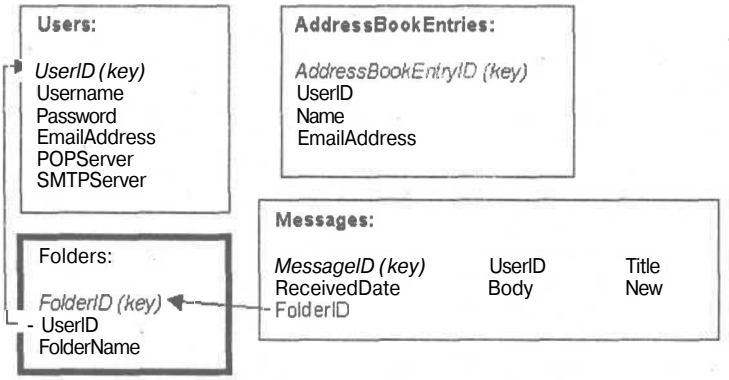

Рис. 7.5. Таблица Folders

В каждой записи таблицы Folders есть ссылка на пользователя через UserID. Для ссылки на папку из записи сообщения таблицы Messagesприменяется идентификатор папки FoldersID.

Нам нужно также позаботиться о файлах, которые могут быть прикреплены к сообщениям. Для этого создана специальная таблица Attachments, поля которой приведены на рис. 7.6.

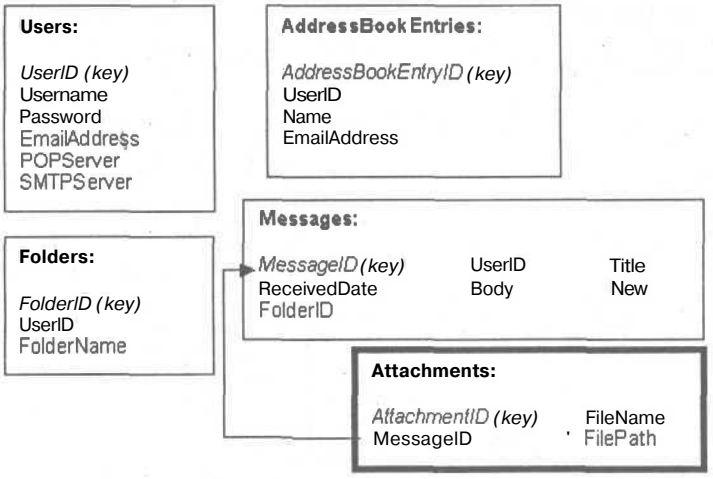

Рис. 7.6. Таблица Attachments

Поля FileName и FilePath указывают местонахождение файла на Peachmail-сервере. Поле MessageID указывает на сообщение в списке Messages, а AttachmentID является идентификатором прикрепленного файла.

На Рисунке 7.7 приведена конечная картинка - схема взаимоотношений объектов (ERD, entity relationship diagram).

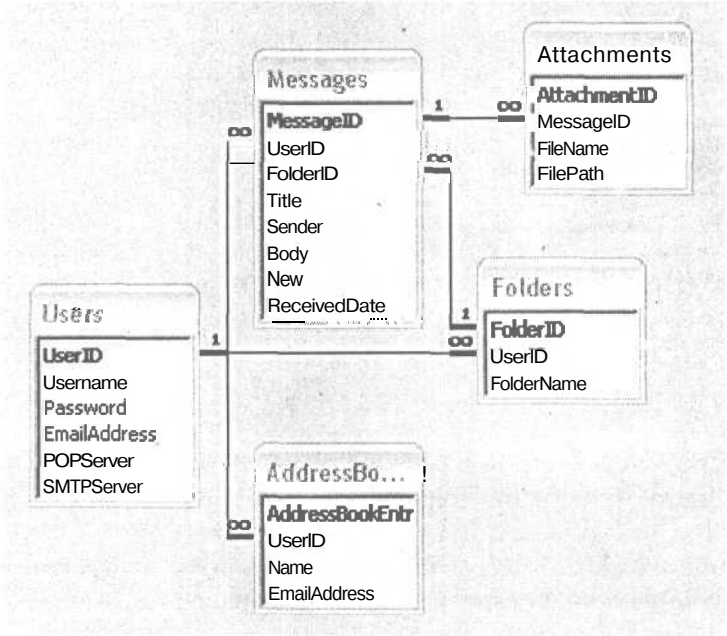

Рис. 7.7. ERD (схема взаимоотношений объектов) данных Peachmail

В табл. 7.1 приведено полное определение данных Peachmail, включая тип каждого поля.

Таблица 7.1. Структуры данных Peachmail

Table Name	Field Name	Field DataType
Users	UserID	AutoNumber
	Username	Text
	Password	Text
	EmailAddress	Text
	POPServer	Text
	SMTPServer	Text
AddressBookEntries	AddressBookEntryID	AutoNumber
	UserID	Number
	Name	Text
	EmailAddress	Text
Messages	MessageID	AutoNumber
	UserID	Number
	FolderID	Number
	Title	Memo
	Body	Memo
	New	Yes/No
	ReceivedDate	Date/Time
Folders	FolderID	AutoNumber
	UserID	Number
	FolderName	Text
Attachments	AttachmentID	AutoNumber
	MessageID	Number
	FileName	Text
	FilePath	Text

Давайте теперь на основе сформулированного определения создадим базу данных.

Написание Peachmail: база данных

Хотя процедура создания базы данных зависит от конкретного типа базы данных, способы решения этой задачи обычно те же самые. Более сложной частью является связывание уже созданной базы данных с сервером. Вначале мы вкратце рассмотрим создание базы данных на основе Microsoft Access, а затем более подробно обсудим связь сервера и базы данных.

Создание базы данных

После загрузки Microsoft Access выберите меню File, и далее - создание новой базы данных. В качестве имени базы данных (рис. 7.8) введите имя peachmail.mdb и нажмите Create.

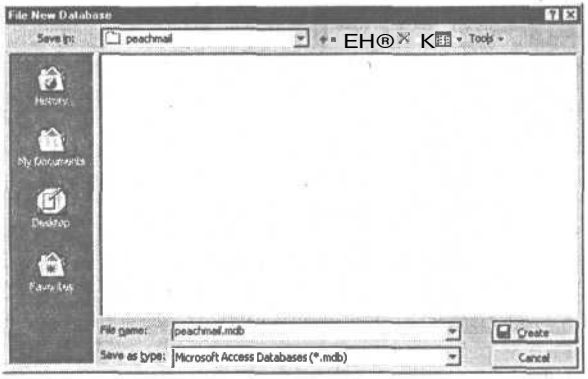

Рис. 7.8. Создание базы данных Microsoft Access

Далее создаем таблицу в Design view, и перед вами появляется пустая форма. Для создания таблиц заполните форму для каждой таблицы, указанной в нашем определении данных. На рис. 7.9 показано, как будет выглядеть таблица Messages.

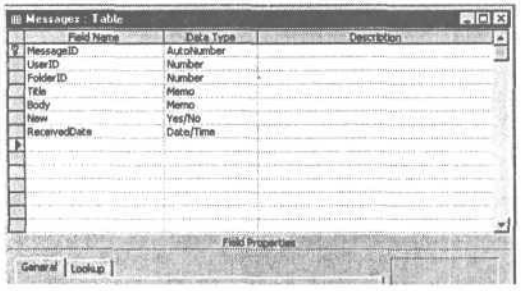

Рис. 7.9. Создание таблицы Messages в Microsoft Access

На этом создание базы данных закончено.

Связывание базы данных с сервером

Чтобы подготовить базу данных к связыванию, создайте с помощью ODBC-отображение (mapping) на веб-хост. Отображение комбинирует директорию и имя файла базы данных на веб-хосте и используется сервером для связи с базой данных. Так как эта книга посвящена Flash, мы не будем обсуждать детали связывания Java с базой данных, отметим только, что в нашем случае требуется драйвер JDBC. Подробную информацию о JDBC можно получить на сайте http://java.sun.com/products/jdbc/.

Написание Peachmail: регистрация и вход под именем пользователя

При работе с Peachmail первым действием Джо была регистрация. Перед ним появилось окно для ввода имени пользователя, пароля, электронного адреса и POP/SMTP-серверов. После ввода этой информации Peachmail отправляет ее на сервер. Затем, после подтверждения (от Peachmail) о создании регистрационной записи, Джо вхо-

дит под своим именем пользователя. Давайте сначала рассмотрим, как Flash MX создает данные и посылает их на сервер (Java), а потом как Flash обрабатывает ответы с сервера (Java).

Клиент: среда Flash MX

В первом окне Peachmail (рис. 7.10) находится две кнопки: Login и New User.

Джо нажимает на кнопку New User (новый пользователь) и переходит к окну регистрации. Он заполняет все детали и нажимает кнопку Create, после чего Flash-клиент создает XML-данные для транзакции CreateUser.

```
<Request>
   <TransactionType>
      CreateUser
   </TransactionType>
   <Data>
      <Username>
         myjoe
      </Username>
      <Password>
         myjoepassword
      </Password>
      <EmailAddress>
         joe@joes_email.com
      </EmailAddress>
      <POPServer>
         mail.joes_email.com
      </POPServer>
      <SMTPServer>
         smtp.joes email.com
      </SMTPServer>
   </Data>
</Request>
```

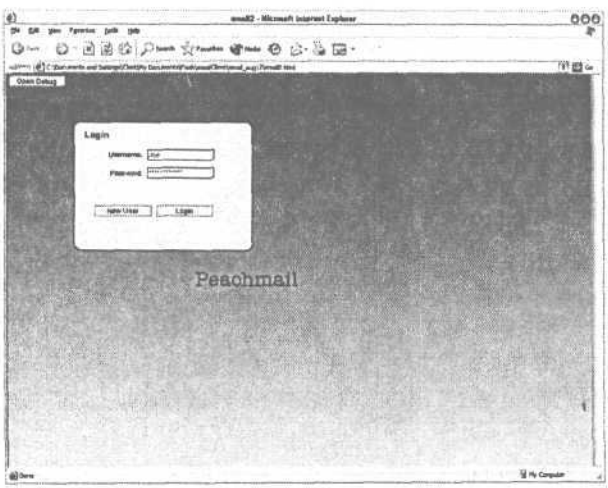

Рис. 7.10. Окно входа в Peachmail

Это вся информация, необходимая Peachmail для регистрации. Затем Peachmail сообщает Джо об успешной регистрации, он возвращается к основному окну и нажимает кнопку Login. После этого Flash создает XML данные транзакции Login.

```
<Request>
   <TransactionType>
      Login
   </TransactionType>
   <Data>
      <Username>
         myjoe
      </Username>
      <Password>
         myjoepassword
      </Password>
   </Data>
</Request>
```

Функции, соответствующие кнопкам Create и Login, называются в Peachmail createHandler и loginHandler. Они участвуют в создании вышеупомянутых XML-документов и передают их на сервер.

Функция createHandler выполняет следующие действия:

1. Создает XML-запрос о регистрации нового пользователя на основе информации из заполненной формы.
2. Передает данный запрос на сервер.
3. Создает функцию обратного вызова (callback) для тестирования, прошла ли регистрация успешно.

Ниже приведен ActionScript-код функции createHandler:

```
createHandler = function () {
   message.text = "Creating...";
   // Проверяем совпадение двух паролей
   if (password.text === passwordRepeat.text) {
      // Создаем XML
      var createUserXml = buildCreateUserRequest(username.text, password.text, email.text, popS
      erver.text, smtpServer.text);
      // Создаем функцию обратного вызова
      createUserCallback = function () {
      if (wasSuccessful(this.getDocIn())) {
         // Переходим к интерфейсу created
         gotoAndStop("created");
      } else {
      // Показываем сообщение
      message.text = getError(this.getDocIn());
   }
}
```

```
        // Отправляем create user XML
        sendToServer(createUserXml, createUserCallback);
        } else {
            // Предупреждаем о неудачной проверке пароля
        message.text = "Passwords do not match";
    }
}
```

Функция loginHandler выполняет следующие действия:

1. Создает XML-запрос о входе под именем пользователя на основе имени пользователя и пароля.
2. Передает данный запрос на сервер.
3. Создает функцию обратного вызова (callback) для проверки успешного входа в систему.

Ниже приведен ActionScript-код функции loginHandler:

```
loginHandler = function () {
    // Создаем XML
    var loginXML = buildLoginRequest(username.text, password.text);
    // Показываем сообщение
    message.text = "Logging In...";
    // Проверяем полученный XML
    loginCallback = function () {
    if (wasSuccessful(this.getDocIn())) {
        // Переходим к интерфейсу email
        gotoAndStop("email");
    } else {
    // Показываем ошибку
        message.text = getError(this.getDocIn());
        }
    }
    // Отправляем запрос на вход в систему
    sendToServer(loginXML, loginCallback);
}
```

Обратите внимание, что функции loginHandler и createHandler ведут себя похоже: обе создают XML-запрос, передают этот запрос на сервер и создают функцию обратного вызова (callback) для проверки ответа с сервера. Такая последовательность действий - create-send-callback (создать, отправить и вызвать callback) - также применяется и в других обработчиках, используемых в Peachmail.

buildBaseRequest: создание XML-запроса

Функция loginHandler вызывает buildLoginRequest, а createHandler вызывает buildCreateUserRequest. Давайте взглянем на соответствующий код:

```
buildLoginRequest = function (username, password) {
    // Создаем запрос
```

```
    var createLoginRequest = new buildBaserequest("Login");
    // Добавляем имя пользователя
    createLoginRequest.addData("Username", username);
    // Добавляем пароль
    createLoginRequest.addData("Password", password);
    // Возвращаем законченный XML документ
    return createLoginRequest.getDoc();
}
buildCreateUserRequest = function (username, password, emailAddress, POPServer, SMTPServer) {
    // Создаем запрос
    var createCreateUserRequest = new buildBaseRequest("CreateUser" );
    // Добавляем имя пользователя
    createCreateUserRequest.addData("Username", username);
    // Добавляем пароль
    createCreateUserRequest.addData("Password", password);
    // Добавляем электронный адрес
    createCreateUserRequest.addData("EmailAddress", emailAddress);
    // Добавляем сервера
    createCreateUserRequest.addData("POPServer", POPServer);
    createCreateUserRequest.addData("SMTPServer", SMTPServer);
    // Возвращаем законченный XML-документ
    return createCreateUserRequest.getDoc();
}
```

Как вы видите, обе функции начинают с создания базового XML-документа с помощью функции buildBaseRequest. Далее XML-документ модифицируется с помощью метода addData (в соответствии с синтаксисом транзакций), и в конце возвращается.

Ниже приведен конструктор buildBaseRequest и метод addData, наряду с некоторыми вспомогательными функциям.

```
// Конструктор buildBaseRequest
function buildBaseRequest(transactionType) {
    //СоздаемXML-объект
    this.doc = new XML();
    // Создаем узел запроса
    var requestNode = this.doc.createElement("Request");
    // Создаем узел транзакции
    var transactionNode = this.doc.createElement("TransactionType");
    // Создаем содержимое транзакции
    var transactionNodeValue = this.doc.createTextNode(transactionType);
    // Добавляем содержимое к узлу транзакции
    transactionNode.appendChild(transactionNodeValue);
    // Создаем узел данных
    var dataNode = this.doc.createElement("Data");
    // Добавляем узел транзакции к узлу запроса
    requestNode.appendChild(transactionNode);
```

```
    // Добавляем узел данных к узлу запроса
    requestNode.appendChild(dataNode);
    // Добавляем узел запроса к XML-документу
    this.doc.appendChild(requestNode);
    // Возвращаем законченный документ
    return this.doc;
} // Конец функции buildBaseRequest
// Функция addData
buildBaseRequest.prototype.addData = function (newData, value, attributes) {
    // Ищем узел данных
    var dataNode = findDataNode(this.doc);
    // Создаем узел newData
    var newDataNode = this.doc.createElement( newData);
    // Проверяем, задано ли содержимое
    if (value != null)
        // Создаем текстовый узел
        var valueTextNode = this.doc.createTextNode(value);
    // Проверяем, указаны ли атрибуты
    if (attributes != undefined)
        // Проходим в цикле по всем атрибутам и задаем их
        for (var i in attributes)
        // Добавляем атрибут
        newDataNode.attributes[i] = attributes[i];
    // Добавляем valueTextNode к newDataNode
    newDataNode.appendChild(valueTextNode);
    // Добавляем newDataNode к dataNode
    dataNode.appendChild(newDataNode);
} // Конец функции addData
// Функция findDataNode
// Эта вспомогательная функция ищет и возвращает
// узел данных XML документа
function findDataNode(xmlDoc) {
    // Извлекаем дочерние узлы
    var children = xmlDoc.firstChild.childNodes;
    // Извлекаем второй дочерний узел (узел данных)
    var dataNode = children[1];
    // Возвращаем узел
    return dataNode;
} // Конец функции findDataNode
// Функция getDoc
// Возвращает XML-документ
buildBaseRequest.prototype.getDoc = function () {
    // Возвращаем XML-документ
    return this.doc;
}
// Функция wasSuccessful
```

```
// Используется для определения успеха транзакции
function wasSuccessful(xmlDoc) {
    // Для надежности создаем XML-объект
    var xmlDoc = new XML(xmlDoc);
    // Разбираем документ в поисках статуса
        var mainNodes = xmlDoc.firstChild.childNodes;
        var statusNode = mainNodes[0];
        var status = statusNode.firstChild.nodeValue;
// В случае ошибки проводим поиск сообщения
if(status == 'Error') {
    var dataNode = mainNodes[1 ];
    var messageNode = dataNode.firstChild;
    errorMessage = messageNode.firstChild.nodeValue;
    return false;
}
// В случае успеха возвращаем true
return true;
} // end wasSuccessful function
// Функция getError
// используется для получения ошибки из XML-документа
function getError(xmlDoc) {
// Для надежности создаем XML-объект
    var xmlDoc = new XML(xmlDoc);
    // Разбираем документ в поисках ошибки
    var mainNodes = xmlDoc.firstChild.childNodes;
    var dataNode = mainNodes[1];
    var messageNode = dataNode.firstChild;
    errorMessage = messageNode.firstChild.nodeValue;
    // Возвращаем ошибку
    return errorMessage;
} // Конец функции getError
```

sendToServer: отправка XML-документа на Java-сервер

Обе функции, loginHandler и createHandler, используют функцию sendToServer() для отправки XML-документа на сервер:

```
sendToServer(loginXML, loginCallback);
```

Ниже приведен код этой функции.

```
global.sendToServer = function (theXML, callBack) {
    // Зададим документ для отправки на сервер
    server.setDocOut(theXML);
    // Зададим документ для получения с сервера
    server.setDocIn(new XML());
    // Извлечем аргументы обратного вызова
    server.callBackArgs = arguments.slice(2);
    // Сохраним функцию обратного вызова
```

```
    server.callBack = callBack;
    // Зададим функцию обратного вызова
    server.onLoad = function (success) {
        // Проверим, получили ли мы документ
        if (success) {
            // Вызываем функцию обратного вызова с соответствующими аргументами
            this.callBack.apply(this, this.callBackArgs);
        } else {
            // Отсоединяемся от сервера
            onServerDisconnect();
        }
    }
    // Посылаем документ
    server.execute();
}
```

Вкратце, функция sendToServer() в качестве параметра использует объект под названием server. Она передает этому объекту XML-данные и функцию обратного вызова (callback), а затем отправляет эти данные с помощью метода execute() объекта server. Объект server относится к классу Serverdata() - созданному специально для передачи данных на сервер (Java). Ниже приведен код, показывающий, как создается объект server и задаются его значения.

```
global.server = new ServerData();
// Задаем метод взаимодействия с сервером
server.setMethod(server.SEND_AND_LOAD);
// Задаем язык
server.setLanguage(server.JAVA);
// Задаем URL-сервера
server.setURL(myurl);
```

Не вникая во все подробности ServerData(), давайте рассмотрим, как метод ServerData.execute() отправляет XML-данные на сервер с помощью Flash метода XML.sendAndLoad(). Соответствующая строчка кода:

```
docOut.sendAndLoad(url, docIn);
```

где docOut - XML-данные, посылаемые на Java-сервер (находящийся по адресу, указанному в переменной url), а docIn будет содержать ответ, возвращаемый сервером после обработки данных.

После получения ответа с сервера Flash автоматически вызывает функцию docIn.onLoad(), которая была заранее установлена в ServerData() на вызов функции обратного вызова (callback).

Функции обратного вызова: контроль возвращаемых результатов

Давайте еще раз вернемся к коду функции loginHandler, а именно к объявлению функции обратного вызова и передаче этой функции объекту ServerData().

```
loginCallback = function () {
```

```
    if (wasSuccessful(this.getDocIn())) {
        // Переходим к интерфейсу email
        gotoAndStop("email");
    } else {
        // Сообщаем об ошибке
        message.text = getError(this.getDocIn());
    }
}
// Посылаем запрос login
sendToServer(loginXML, loginCallback);
```

Функция loginCallback передается как второй параметр функции sendToServer, так что объект ServerData может автоматически вызвать loginCallback() после получения результатов обработки XML на Java-сервере. Также обратите внимание, что в случае успешной транзакции loginCallback переводит пользователя к основному окну программы (окно почты). В случае неудачи сообщается о неудачной транзакции.

Теперь, когда мы рассмотрели последовательность действий create-send-callback (создать, отправить, вызвать callback), давайте перейдем к серверу и обсудим, как он получает данные, обрабатывает их и возвращает результаты Flash-клиенту.

Сервер: Java

Вспомните, как Flash отправляет XML-данные на сервер - используется объект ServerData() и стандартный метод sendAndLoad(), отправляющий XML (из docOut) на сервер:

```
docOut.sendAndLoad(url, docIn);
```

Также вспомните о том, что переменная docIn содержит ответ с сервера. Как это работает? При вызове метода sendAndLoad() Flash будет ждать до получения ответа (сохраняя его в переменной docIn). После сохранения результатов в переменной docIn Flash автоматически вызывает метод docIn.onLoad().

Вызов docIn.onLoad() приводит к вызову callback-функции, так как метод docIn.onLoad() был настроен на вызов callback-функции при вызове функции sendToServer():

```
sendToServer(loginXML, loginCallback);
```

Обработка запроса NewUser

Вы видели раньше, как Flash создает запрос New User при нажатии Джо в окне регистрации кнопки Create. Когда Java-сервер получает эти данные и обрабатывает их, он сообщает Flash-клиенту об успехе или неудаче. Как обсуждалось ранее, ответ сохраняется во Flash-переменной docIn.

В случае неудачи Java-сервер возвращает следующее:

```
<Response>
    <Status>
```

```
        Error
      </Status>
      <Data>
        <Message>
          Сообщение о причине неудачи запроса
        </Message>
      </Data>
    </Response>
```

а в случае успеха:

```
<Response>
  <Status>
    Success
  </Status>
  <Data>
    <Message>
        The transaction completed successfully
    </Message>
  </Data>
</Response>
```

Обработка запроса Login

Так же, как и в случае обработки запроса о регистрации нового пользователя, в случае неудачи при обработке Login запроса возвращается следующее:

```
<Response>
  <Status>
    Error
  </Status>
  <Data>
    <Message>
      Сообщение о причине неудачи запроса
    </Message>
  </Data>
</Response>
```

а в случае успеха:

```
<Response>
  <Status>
    Success
  </Status>
  <Data>
    <Message>
      The transaction completed successfully
    </Message>
  </Data>
</Response>
```

База данных

При обработке запроса о регистрации нового пользователя в базе данных обновляется таблица Users (Java-сервер посылает информацию о новом пользователе в базе данных).

При обработке запроса Login (вход под именем пользователя) Java-сервер создает новый поток (thread) для работы с Джо до тех пор, пока он не выйдет из системы. База данных в этом случае не меняется.

Написание Peachmail: услуги адресной книги

Теперь, когда Джо вошел под своим именем пользователя, ему нужно добавить в адресную книгу адреса нескольких друзей. Адресная книга Peachmail дает возможность пользователю добавить контакты, удалить их и отправить им сообщение. Давайте рассмотрим, как Flash MX работает вместе с сервером при помощи адресной книги.

Клиент: Flash MX

Когда Джо входит под именем пользователя, перед ним появляется основное окно электронной почты (рис. 7.11).

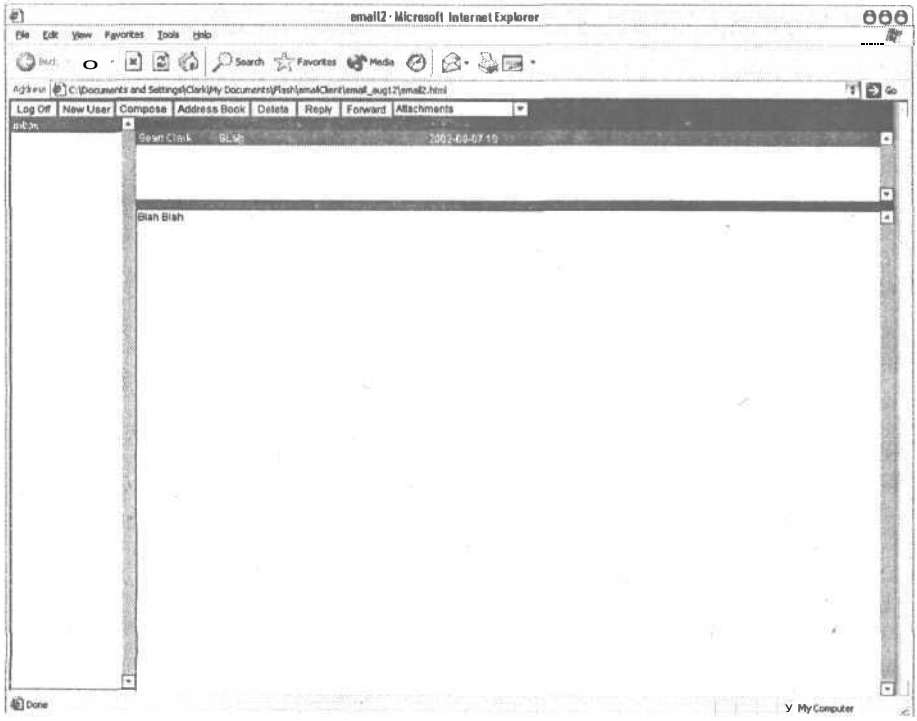

Рис. 7.11. Основное окно электронной почты Peachmail

Каждая кнопка на экране ведет себя так же, как и кнопки в окнах регистрации и входа в систему, - при нажатии выполняется функция-обработчик. Так как Джо хочет добавить в адресную книгу нескольких друзей, он нажимает кнопку Address Book. Это приводит к вызову функции addressBookHandler, которая просто открывает всплывающее окно для показа адресной книги:

```
global.openBrowserWindow = function (url, name, width, height) {
    getURL("javascript:void(window.open('" + url + "', '" + name + "', 'resizable=0, toolbar=0,
    menubar=0, titlebar=0, status=0, width=" + width + ", height=" + height + "'))");
    }
addressBookHandler = function () {
    openBrowserWindow("addressBook.html", "addressBook", 217, 157);
}
```

В окне адресной книги (Address Book, рис. 7.12), находятся три кнопки, позволяющие Джо добавить новый адрес, удалить адрес или послать сообщение по выделенному адресу.

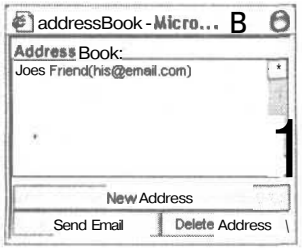

Рис. 7.12. Добавить новый адрес, удалить адрес или послать сообщение в окне Address Book

Давайте рассмотрим каждую из этих услуг отдельно.

Добавление нового адреса

При нажатии кнопки New Address вызывается функция newAddressHandler(), открывающая окно New Address (рис. 7.13) для добавления нового адреса:

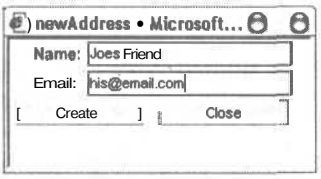

Рис. 7.13. Окно New Address

При загрузке окна New Address создается локальное соединение с окном Address Book для передачи добавляемого адреса. Локальные соединения позволяют передавать данные между двумя внешними (external) .swf-файлами. В конце главы мы обсудим локальные соединения более детально, а сейчас давайте посмотрим, что происходит при нажатии кнопки Create:

```
createHandler = function () {
    // Передаем адрес
    newAddressConnection.send("newAddress", name.text, email.text);
    // Закрываем окно
```

```
    getURL("javascript:void(window.close());");
}
```

Этот обработчик берет значения текстовых полей name и email, отправляет их в функцию локального соединения newAddress() - локальное соединение при этом является окном Address Book, - а затем закрывает себя (окно New Address). Код функции newAddress() приведен ниже.

```
newAddressConnection.newAddress = function (name, email) {
    // Создаем новый адрес
    addAddress(name, email);
}
```

Функция newAddress() берет имя, электронный адрес и вызывает с ними функцию addAddress():

```
addAddress = function (name, email) {
    // Создаем XML
    addAddressXml = buildAddAddressBookEntryRequest(name, email);
    // Создаем callback
    addAddressCallback = function (name, email) {
    // Проверяем на успех
    if (wasSuccessful(this.getDocIn())) {
        // Извлекаем dataNode
        var dataNode = findDataNode(this.getDocIn());
        // Получаем id
        var id = dataNode.firstChild.attributes.ID;
        // Добавляем в адресную книгу
        addressBook.addAddress(name, email, id);
    }
}
// Передаем на сервер (имя и адрес в качестве параметров)
sendToServer(addAddressXml, addAddressCallback, name, email);
}
```

Здесь функция buildAddAddressBookEntryRequest() создает новый XML-запрос, а функция sendToServer() передает этот запрос для обработки на сервер. Как вы помните, это похоже на последовательность действий create-send-callback (создать, отправить, вызвать callback), рассмотренную нами в разделе регистрации и входа в систему. Здесь нужно обратить внимание на callback-функцию: она берет (в качестве параметра) XML ответ с сервера, разбирает его, а затем вызывает addressBook.addAddress() для добавления деталей нового адреса в компонент списка Address Book (listbox):

```
addressBook.prototype.addAddress = function (name, email, id) {
    // Создаем временный объект
    var tempObject = {};
    // Сохраняем имя
    tempObject.name = name;
    // Сохраняем email
```

```
    tempObject.email = email;
    // Сохраняем id
    tempObject.id = id;
    // Добавляем в массив addresses
    this.addresses.push(tempObject);
    // Добавляем в listbox
    this.listbox.addItem(tempObject.name + "(" + tempObject.email + ")", this.address.length);
    // Сохраняем в объекте ids
    this.ids[id] = this.listbox.getLength() - 1;
}
```

Заметьте, как в коде для добавления нового адреса вызывается метод addItem() Macromedia компонента listbox.

Удаление адреса

Для удаления адреса его нужно выделить в компоненте списка (listbox), а затем нажать кнопку Delete Address. После выделения элемента в окне списка (listbox) его индекс можно получить с помощью функции listbox.getSelectedIndex(). При нажатии кнопки Delete Address выполняется следующий код:

```
deleteAddressHandler = function () {
// Получим id выбранного адреса
var id = addressBook.getSelectedAddress().id;
// Удалим адрес
deleteAddress(id);
}
```

Функция addressBook.getSelectedAddress() возвращает запись, содержащую имя, текст и идентификатор (id) выделенного адреса, с помощью упомянутой ранее функции listbox.getSelectedIndex().

```
addressBook.prototype.getSelectedAddress = function () {
    // Получим выбранный элемент
    var selected = this.listbox.getSelectedIndex();
    // Возвращаем объект с информацией о выбранном элементе
    return this.addresses[selected];
}
```

Функция deleteAddress() создает XML-запрос на удаление адреса, отправляет его на сервер с помощью sendToServer() и затем удаляет выбранный адрес из адресной книги:

```
deleteAddress = function (id) {
    // Создаем xml
    deleteAddressXml = buildDeleteAddressBookEntryRequest(id);
    // Отправляем xml
    sendToServer(deleteAddressXml);
    // Удаляем из адресной книги
    addressBook.deleteAddress(id);
}
```

Конечным шагом функции deleteAddress является вызов addressBook.deleteAddress(), удаляющий адрес из компонента списка (listbox) Address Book:

```
addressBook.prototype.deleteAddress = function (id) {
    // Получаем индекс из объекта ids
    var index = this.ids[id];
    // Удаляем из listbox
    this.listbox.removeItemAt(index);
}
```

Отправка сообщения (email)

Для отправки сообщения пользователю сначала нужно выделить адрес в окне Address Book, а затем нажать кнопку Send Email.

```
sendEmailHandler = function () {
    // Получаем выделенный адрес
    var email = addressBook.getSelectedAddress().email;
    // Открываем окно для написания сообщения
    openCompose(email);
}
```

Функция sendEmailHandler() передает выделенный адрес функции openCompose(), которая открывает окно Compose с выделенным адресом и создает локальное соединение с этим окном.

```
openCompose = function (to, subject, message) {
    // Создаем соединение с окном compose
    composeConnection = new popupConnection("compose", false);
    // Задаем содержимое поля email
    composeConnection.send("setVariable", "to", "text", to);
    // задаем содержимое поля заголовка
    composeConnection.send("setVariable", "subject", "text", subject);
    // Задаем текст сообщения
    composeConnection.send("setVariable", "message", "text", message);
    // Функция для закрытия соединения после того, как все отправлено
    composeConnection.onFinishQueue = function () {
        // Закрываем соединение
        this.close();
    }
    // Открываем окно compose
    openBrowserWindow("compose.html", "compose", 550, 400);
}
```

Появляется окно Compose (рис. 7.14) и пользователь может заполнить текстовые поля, а затем нажать Send (или Cancel, для отмены).

Рис. 7.14. Окно Compose программы Peachmail

Обработчик кнопки Send выглядит следующим образом:

```
sendHandler = function () {
    // Создаем xml
    var sendMsgXml = buildSendMessageRequest(to.text, subject.text, message.text);
    // Создаем callback
    sendMsgCallback = function () {
        // Закрываем окно
        getURL("javascript: void(window.close());");
    }
    // Отправляем созданный XML на сервер
    sendToServer(sendMsgXml, sendMsgCallback);
}
```

Как видите, этот обработчик только создает XML-запрос и отправляет его на сервер, после чего callback закрывает окно. Сервер сам отправляет сообщение по указанному адресу.

Сервер: Java

Аналогично обработке запросов на регистрацию и вход в систему обработка сервером запросов, связанных с адресной книгой, также состоит из получения XML-документа, выполнения соответствующего запроса и возвращения результата (успех или неудача). При добавлении адреса также происходит добавление адреса в базу данных. При удалении адрес удаляется из базы данных. И наконец, при отправке сообщения сервер отвечает за отправку сообщения по указанному адресу.

Обработка запроса на добавление адреса

Когда Джо добавляет новый адрес в адресную книгу, на сервер посылается примерно следующее:

```
<Request>
    <TransactionType>AddAddressBookEntry</TransactionType>
```

```
    <Data>
      <Entry>
        <Name>Lara</Name>
      <Email>lara@iamlara.com</Email>
    </Entry>
    </Data>
</Request>
```

В случае успешного добавления адреса в базу данных сервер возвращает.

```
<Response>
    <Status>Success</Status>
    <Data>
      <Entry ID="456" />
    </Data>
</Response>
```

а в случае неудачи-

```
<Response>
    <Status>Error</Status>
    <Data>
      <Message>The address did not add successfully</Message>
    </Data>
</Response>
```

Обработка запроса на удаление адреса

Предположим, что Джо успешно добавил своего друга в адресную книгу, но потом сообразил, что ошибся в адресе, и решает удалить данный адрес. XML-запрос будет выглядеть примерно следующим образом:

```
<Request>
    <TransactionType>DeleteAddressBookEntry</TransactionType>
    <Data>
      <Entry ID="56" />
    </Data>
</Request>
```

В случае успешного удаления сервер вернет:

```
<Response>
    <Status>Success</Status>
    <Data>
      <Message>The transaction completed successfully</Message>
    </Data>
</Response>
```

а в случае неудачи-

```
<Response>
    <Status>Error</Status>
```

```
    <Data>
      <Message>The address did not delete successfully</Message>
    </Data>
  </Response>
```

Обработка запроса на отправку сообщения

Когда Джо решает отправить сообщение, XML-запрос будет выглядеть как:

```
<Request>
  <TransactionType>SendMsg</TransactionType>
  <Data>
    <Addressee>lara@iamlara.com</Addressee>
    <Subject>curious</Subject>
    <Body>Lara, do you miss me? Curious, Joe</Body>
  </Data>
</Request>
```

В случае успешной отправки сообщения сервер возвращает следующий XML-ответ:

```
<Response>
  <Status>Success</Status>
  <Data>
    <Message>The transaction completed successfully</Message>
  </Data>
</Response>
```

а в случае неудачи-

```
<Response>
  <Status>Error</Status>
  <Data>
    <Message>The email failed to send</Message>
  </Data>
</Response>
```

База данных

При добавлении адреса сервер добавляет в таблицу AddressBookEntries новую запись, содержащую идентификатор Джо (userID), имя нового человека и электронный адрес. При удалении адреса из таблицы AddressBookEntries удаляется соответствующая запись. И наконец, при запросе на отправку сообщения это сообщение сохраняется в таблице Messages.

Написание Peachmail: услуги электронной почты

Джо закончил с добавлением своих друзей в адресную книгу и готов воспользоваться услугами электронной почты Peachmail! Эти услуги позволяют просматривать, посылать, получать и удалять электронную почту. Ранее уже было

показано, как работает отправка сообщения из адресной книги (окно Address Book). Вы увидите, что отправка сообщения из основного окна отличается нена много.

Клиент: Flash MX

В предыдущих разделах мы рассмотрели кнопки основного окна (рис. 7.15) New User и Address Book. Теперь давайте рассмотрим оставшиеся кнопки: Compose, Delete, Reply и Forward.

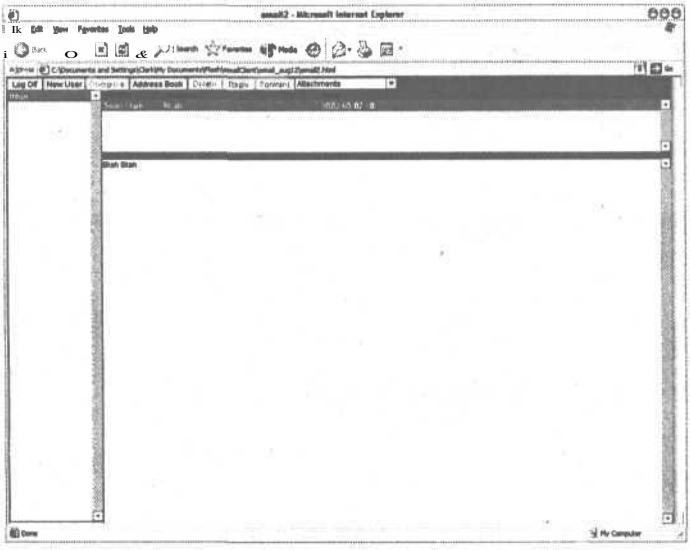

Рис. **7.15.** Основное окно Peachmail

Отправка почты

Все три кнопки, Compose, Reply и Forward, ведут к отправке сообщения. Они отличаются только тем, как создается всплывающее окно Compose. Взгляните на соответствующие обработчики и заметьте, как они используют метод openCompose():

```
composeHandler = function (} {
    // Открываем окно compose
    openCompose();
}
replyHandler = function () {
    // Получаем выделенный адрес
    var email = emailMessages.getSelectedEmail();
    // Получаем заголовок и добавляем "RE: "
    var subject = "RE: " + email.emailObject.subject;
    // Получаем from адрес
    var fromAddress = email.email;
```

```
    // Форматируем сообщение
    var message = "\n--- Original Message ---\n" + addTicks(emailText.text);
    // Открываем окно compose и задаем переменные
    openCompose(fromAddress, subject, message);
}
forwardHandler = function () {
    // Получаем выделенный адрес
    var email = emailMessages.getSelectedEmail();
    // Получаем заголовок и добавляем "FW: "
    var subject = "FW: " + email.emailObject.subject;
    // Получаем сообщение
    var message = "\n--- Original Message ---\n" + addTicks(emailText.text);
    // Открываем окно compose с заданными переменными
    openCompose("", subject, message);
}
```

Здесь метод openCompose() тот же самый, что и в окне Address Book. В общем, каждый вызов openCompose() заполняет определенные поля в окне Compose (в зависимости от вызывающего обработчика). Например, нажатие кнопки Reply (ответ) заполняет в окне Compose поля "To", заголовка и тела сообщения данными из выбранного в основном окне сообщения: отправитель, заголовок и само сообщение. Нажатие Forward заполняет в окне Compose поля заголовка и тела сообщения (на основе выбранного сообщения), а нажатие Compose просто открывает окно Compose.

Обратите внимание, что обработчики кнопок Forward и Reply требуют наличия выделенного сообщения в основном окне (для заполнения окна Compose соответствующим содержимым). Это относится к строчке

```
var email = emailMessages.getSelectedEmail();
```

Объект emailMessages относится к специальному классу emailList, используемому для управления обработкой сообщений. Мы обсудим этот класс более подробно в конце главы. А сейчас можно показать, что метод этого класса getSelectedEmail() вызывает метод getSelectedItem() компонента списка (listbox) для получения выделенного сообщения:

```
emailList.prototype.getSelectedEmail = function () {
    // Возвращаем id
    return this.listBox.getSelectedItem().data;
}
```

Переменная data является объектом под названием emailObject, в котором находится содержимое сообщения, т. е. email.sender, email.email, email.subject, mail.ReceiveDate и так далее.

Надо также указать, что тело сообщения извлекается в строчке

```
addTicks(emailText.text);
```

emailText - это текстовое поле, показывающее тело выделенного сообщения, а функция addTicks() просто добавляет к каждой строчке сообщения метки, чтобы обозначить, что они относятся к старому сообщению.

После открытия окна Compose все происходит точно также, как и при отправке сообщения из адресной книги (описанной ранее).

Получение почты

Peachmail сохраняет полученные сообщения в папке Inbox. Папка Inbox находится на рабочем поле в компоненте списка (listbox) под названием folders. При щелчке мышью по этой папке Flash посылает серверу запрос на получение сообщений, принадлежащих к данной папке. Для того, чтобы понять, как сообщения извлекаются и отображаются на экране, необходимо сначала понять, как эти сообщения хранятся во Flash.

В Peachmail есть специальный класс под названием folderList():

```
global.folderList = function (listbox) {
    // Сохраняем ссылку на listbox
    this.listbox = listbox;
    // Задаем обработчик onChange
    this.listbox.setChangeHandler("onChange", this);
    // Задаем массив folders
    this.folders = [];
}
```

Данный класс используется для управления всеми папками пользователя. Каждая папка отображается на экране с помощью Mactomedia компонента списка (listbox). Информация о папках сохраняется в массиве this.folders.

Когда пользователь входит в Peachmail, программа сразу же вызывает функцию getFolders() для получения его папок. Так же как и в случае с обработчиками, рассмотренными в предыдущих разделах, эта функция совершает последовательность действий create-send-callback, а именно создает XML-запрос Get Folders, отправляет его на сервер и обрабатывает полученную с сервера информацию о папках.

Ниже приведен ActionScript-код getFolders и callback функции.

```
getFolders = function () {
    // Создаем xml
    var getFoldersXml = buildGetFoldersRequest();
    // Создаем callback
    getFoldersCallBack = function () {
        // Создаем список папок
        global.emailFolders = new folderList(folders);
        //Удалим возможные старые сообщения
        folders.removeAll();
        // Создаем onSelect callback
        emailFolders.onSelect = function (folder) {
```

```
            // Получим id
            var id = folder.data;
            // Получим сообщения
            getMessages(id);
        }
        // Создаем массив folders
        foldersArray = [];
        // Получим узел данных
        var dataNode = findDataNode(this.getDocIn());
        // Получим папки
        var folders = dataNode.firstChild.childNodes;
        // Цикл по всем папкам
        for (var i = 0; i < folders.length; i++) {
            // Получим папку
            var folder = folders[i];
            // Получим id
            var id = folder.attributes.ID;
            // Получим имя
            var name = String(folder.firstChild.firstChild);
            // Добавим папку
            foldersArray.push({id: id, name: name, emails:[]});
            // Добавим в массив папок
            emailFolders.addFolder(name, id);
        }
    }
    // Отправим на сервер
    sendToServer(getFoldersXml, getFoldersCallback);
}
```

Вы видите, что callback-функция getFoldersCallback значительно больше, чем callback функции в обработчиках, рассмотренных ранее, так что давайте рассмотрим ее более подробно.

В самом начале создается объект folderList под названием emailFolders, указывающий на компонент списка folders на рабочем поле:

```
global.emailFolders = new folderList(folders);
```

Затем папки очищаются с помощью метода removeAll() и объявляется метод onSelect объекта emailFolders. Этот метод является основным при получении сообщений, мы обсудим его позже. Далее создается массив foldersArray. Потом callback функция проводит разбор (parsing) полученных данных. Для каждой найденной папки извлекается имя и идентификатор папки, а затем:

1. Найденное имя и идентификатор сохраняются в соответствующей записи в массиве foldersArray, вместе с пустым массивом под названием emails:

```
foldersArray.push({id: id, name: name, emails: []});
```

2. Имя и идентификатор добавляются в объект emailFolders в качестве новой папки:

```
emailFolders.addFolder(name, id);
```

Метод emailFolders.addFolder() обновляет на экране компонент списка folders, чтобы показать добавленную папку, а также обновляет массив folders объекта emailFolders (добавляя запись, содержащую имя папки, идентификатор и индекс соответствующего компонента списка).

Ниже приведен ActionScript-код метода addFolder:

```
folderList.prototype.addFolder = function (name, id) {
    //Добавляем в компонент списка
    this.listbox.addItem(name, id);
    //Сохраняем в массиве
    this.folders.push({name: name, item: this.listbox.getLength(), id: id});
}
```

Теперь мы видим, как обработчик обновляет компонент списка на экране. Однако не совсем очевидно, как этот обработчик связывается с компонентом folders (чтобы при щелчке мышью в компоненте folders вызывался именно данный обработчик). Для этого нужно понять Macromedia компонент списка (listbox) и как он обрабатывает изменения.

Когда пользователь выделяет элемент в компоненте списка, автоматически вызывается обработчик события onChange(). Так как по умолчанию у этого события нет никакого обработчика, то ничего не происходит до тех пор, пока не будет задан обработчик. Вспомните следующую строчку из конструктора класса folderList:

```
this.listbox.setChangeHandler("onChange", this);
```

Когда в getFoldersCallback() был создан объект folderList под названием emailFolders, была выполнена также приведенная выше строчка из конструктора и метод onChange() компонента списка folders был назначен методом onChange класса folderList:

```
folderList.prototype.onChange = function () {
    // Вызываем callback-функцию с выделенной папкой в качестве параметра
    this.onSelect(this.listbox.getSelectedItem());
}
```

Обратите внимание, что метод onChange вызывает метод onSelect. Метод onSelect был задан в getFoldersCallback() и выглядит следующим образом:

```
emailFolders.onSelect = function (folder) {
    // Получаем id
    var id = folder.data;
    // Получаем сообщения
    getMessages(id);
}
```

Этому методу в качестве параметра передается выделенная папка из emailFolders.onChange(). Вначале извлекается идентификатор папки, а затем он передается функции getMessages(). Функция getMessages() выполняет последовательность действий create-send-callback для получения сообщений, относящихся к данной папке. Более всего интересны здесь действия, выполняемые callback-функцией getMessagesCallback() для обработки полученных данных и их отображения на экране.

Ниже приведен ActionScript-код функции getMessages:

```
getMessages = function (folderId) {
    // Создаем xml
    var getMessagesXml = buildGetMessagesInFolderRequest(folderId);
    // Создаем callback
    getMessagesCallback = function () {
        // Проверяем, успешно ли получены данные
        if (wasSuccessful(this.getDocIn())) {
            // Создаем массив сообщений email
            emailMessages = new emailList(messages);
            // Удаляем все сообщения из компонента listbox
            messages.removeAll();
            // Создаем обработчик
            emailMessages.onSelectEmail = function(emailData) {
            // Получаем детали сообщения
            getMessageDetails(emailData.id);
            // Делаем доступными кнопки reply и forward
            reply.setEnabled(true);
            forward.setEnabled(true);
            // Проверяем, нет ли приложенных файлов
            if (emailData.emailobject.hasAttachments == "True") {
                // Делаем доступным окно combo
                attachmentCombo.setEnabled(true);
            } else {
                // Делаем недоступным окно combo
                attachmentCombo.setEnabled(false);
            }
            // Запоминаем ранее выделенный listbox
            global.selectedListBox = "emails";
        }
        // Создаем обработчик удаления сообщения
        emailMessages.onDeleteEmail = function(emailData) {
            // Получаем id
            var id = emailData.id;
            debug.print(":::" + id);
            // Получаем xml
            var deleteEmailXml = buildDeleteMessageRequest(id);
            // Отправляем на сервер
```

```
            sendToServer(deleteEmailXml);
        }
        // Получаем узел данных
        var dataNode = findDataNode(this.getDocIn());
        // Получаем сообщения
        var messages = dataNode.firstChild.childNodes;
        // Цикл по всем сообщениям
        for (var i = 0; i < messages.length; i++) {
            // Получаем сообщение
            var message = messages[i];
            // Создаем временный объект
            var messageObject = {};
            // Получаем id
            messageObject.ID = message.attributes.ID;
            // Получаем статус (было уже прочитано или нет)
            messageObject. New = message.attributes.New;
            // Получаем дату прихода сообщения
            messageObject. ReceiveDate = message.attributes.ReceiveDate;
            // Получаем, есть ли приложенные файлы
            messageObject. HasAttachments = message.attributes.HasAttachments;
            // Извлекаем имя отправителя и адрес
            var sender = message.attributes.Sender.split("(");
            // Получаем имя отправителя
            messageObject.Sender = sender[1].substring(0, sender[1].length - 1);
            // Получаем адрес
            messageObject.email = sender[0];
            // Получаем заголовок
            messageObject.Subject = unescape(message.firstChild.firstChild.toString());
            // Создаем массив для приложенных файлов
            messageObject.attachments = [];
            // Добавляем сообщение в массив email
            emailMessages.addEmail(messageObject);
            }
        }
    }
    // Отправляем xml
    sendtoServer(getMessagesXml, getMessagesCallback);
}
```

Функция обратного вызова getMessagesCallback() задает объект emailList под названием emailMessages на основе компонента messages с рабочего поля:

```
emailMessages = new emailList(messages);
```

Объект emailMessages создан для облегчения работы с сообщениями в данной папке. Затем создаются два обработчика:

```
emailMessages.onSelectEmail()
emailMessages.onDeleteEmail()
```

которые будут вызваны позже, при выделении или удалении пользователем сообщения (в компоненте messages). В конце, функция getMessagesCallback() проходит в цикле по всем полученным сообщениям и добавляет каждое как объект в emailMessages в следующей строчке:

```
emailMessages.addEmail(messageObject);
```

Ниже приведен код функции addEmail(), показывающий, как компонент messages на экране обновляется с помощью метода addItem().

```
emailList.prototype.addEmail = function (email) {
    // Добавляем сообщение
    this.listBox.addItem(email.sender, emaii.subject, email.ReceiveDate, {id: email.id, email: email.email,
    emailObject: email});
    // Получаем длину списка сообщений
    var length = this.listBox.getLength();
    // Сохраняем индекс в объекте ids
    this.ids[email.id] = length - 1;
}
```

При выделении элемента компонента messages автоматически вызывается обработчик onChange(). Так же, как и в конструкторе класса emailFolders, конструктор класса emailList связывает обработчик события onChange() компонента messages с методом onChange(). Ниже приведен код onChange.

```
emailList.prototype.onChange = function () {
    // Вызываем callback-функцию с id в качестве параметра
    this.onSelectEmail(this.listBox.getSelectedItem().data);
}
```

Так что, когда Джо делает щелчок мышью по сообщению в компоненте списка messages, emailList.onChange() вызывает метод onSelectEmail() (заданный ранее), что приводит к появлению сообщения. В этот момент все сообщения уже получены с сервера и готовы для просмотра.

Просмотр почты

Выше было показано, что для обновления окна messages пользователю нужно щелкнуть мышью по папке в окне folders. В этом случае выполняется функция getMessages(), размещающая в окне messages заголовки всех сообщений из данной папки.

Избегая лишних повторений, скажем только, что функция getMessages() очень похожа на getFolders(); она совершает такую же последовательность действий create-send-callback для получения сообщений указанной папки, а затем создает callback функцию для просмотра сообщения (при щелчке по заголовку сообщения). Вспомните, что при щелчке мышью по сообщению вызывается функция onSelectMail(), передающая идентификатор messageID данного сообщения функции getMessageDetails():

```
getMessageDetails = function (messageID) {
    // Создаем xml
```

```
var getMessageDetailsXml = buildGetMessageDetailsRequest(messageID);
// Создаем callback
getMessageDetailsCallback = function (messageID) {
    // Проверяем успешность получения ответа
    if (wasSuccessful(this.getDocIn())) {
        // Получаем узел данных
        var dataNode = findDataNode(this.getDocIn());
        // Получаем сообщение
        var message = dataNode.firstChild;
        // Получаем тело сообщения
        var body = message.childNodes[2];
        // Получаем id
        var emailId = message.attributes.ID;
        // Получаем объект сообщения
        var email = emailMessages.getEmail(emailId).emailObject;
        // Проверяем, есть ли приложенные файлы
        if (email.hasAttachments == "True") {
            // Получаем приложенные файлы
            var attachments = message.childNodes[3].childNodes;
            // Создаем массив для хранения приложенных файлов
            var attachmentsArray = [];
            // Цикл по всем приложенным файлам
            for (var i = 0; i < attachments.length; i++) {
                // Получаем id
                var id = attachments[i].attributes.ID;
                // Получаем имя
                var name = attachments[i].firstChild.firstChild.toString();
                // Добавляем в массив
                attachmentsArray.push({name: name, id: id});
            }
            // Обрабатываем приложенные файлы
            processAttachments(attachmentsArray);
        }
        // Задаем текст
        emailText.text = removeTags(body);
    }
}
    // Отправляем XML на сервер
    sendToServer(getMessageDetailsXml,
getMessageDetailsCallback,  messageID);
}
```

Функция getMessageDetails() также использует последовательность действий create-send-callback для получения сообщения с сервера, помещая текст тела сообщения в текстовое поле на рабочем поле.

Удаление сообщения

Удаление сообщения требует двух щелчков мышью: вначале пользователь щелкает по сообщению в окне messages, а затем нажимает кнопку Delete. Давайте начнем с более легкой части и рассмотрим обработчик кнопки Delete.

```
deleteHandler = function () {
    // Удаляем выделенное сообщение
    emailMessages.deleteSelectedEmail();
}
```

Принимая во внимание, что emailMessages является объектом класса emailList(), функция deleteSelectedEmail() выглядит следующим образом:

```
emailList.prototype.deleteSelectedEmail = function () {
    // Получаем выделенное сообщение
    var email = this.listBox.getSelectedItem().data;
    // Получаем номер выделенного сообщения
    var num = this.ids[email.id];
    // Удаляем из компонента списка (listbox)
    this.listBox.removeItemAt(num);
    // Вызываем callback
    this.onDeleteEmail(email);
}
```

Здесь сначала сообщение удаляется из окна messages, а потом вызывается onDeleteEmail(). Для удобства приведем еще раз код метода onDeleteEmail():

```
emailMessages.onDeleteEmail = function (emailData) {
    // Получаем id
    var id = emailData.id;
    // Получаем xml
    var deleteEmailXml = buildDeleteMessageRequest(id);
    // Посылаем на сервер
    sendToServer(deleteEmailXml);
}
```

Обратите внимание, что onDeleteEmail() передает функции sendToServer() только один аргумент: XML-запрос на удаление записи из базы данных. В этом случае callback не требуется, так как сервер просто передает запрос на удаление записи в базу данных.

Давайте теперь перейдем к рассмотрению состава некоторых транзакций между Flash и сервером, связанных с почтовыми услугами.

Сервер: Java

Почтовые услуги используют сервер при отправке, получении, просмотре и удалении сообщения. Далее следует краткий обзор соответствующих транзакций между Flash и сервером.

Запрос на отправку сообщения

XML-запрос и ответ в этом случае совпадают с описанными ранее в разделе услуг адресной книги.

Запрос на получение

Этот запрос разбит на две стадии: запрос данных о папках и запрос о сообщениях (из выделенной папки). Ниже приведен пример запроса Джо на получение данных о папках.

```
<Request>
   <TransactionType>GetFolders</TransactionType>
   <Data></Data>
</Request>
```

А здесь приведен пример успешного ответа:

```
<Response>
   <Status>Success</Status>
   <Data>
      <Folders>
         <Folder ID="3">
         <Name>Inbox</Name>
      </Folder>
      </Folders>
   </Data>
</Response>
```

Предполагая, что Джо получил успешный ответ, он далее щелкает мышью по появившейся папке, посылая запрос на сообщения:

```
<Request>
   <TransactionType>GetMessagesInFolder</TransactionType>
   <Data>
      <Folder ID="3"/>
   </Data>
</Request>
```

Пример успешного ответа:

```
<Response>
   <Status>Success</Status>
   <Data>
      <Messages>
         <Message ID="5" New="True" ReceiveDate="2002-09-18" HasAttachments="False">
            <Subject>Hello there Joe!</Subject>
         </Message>
      </Messages>
   </Data>
</Response>
```

Запрос на просмотр конкретного сообщения

Когда Джо решает посмотреть какое-нибудь сообщение, щелчок мышью по данному сообщению (в соответствующем окне сообщений) приводит к отправке на сервер запроса Message:

```
<Request>
   <TransactionType>GetMessageDetails</TransactionType>
   <Data>
      <Message ID="5"/>
   </Data>
</Request>
```

В случае успеха возвращается следующее:

```
<Response>
   <Status>Success</Status>
   <Data>
      <Message ID="5" ReceiveDate="2002-09-18">
         <Subject>Hello there Joe!</Subject>
         <Body>Joe! You're the best! Love, Lara</Body>
         <Attachments>
      </Attachments>
      </Message>
   </Data>
</Response>
```

Запрос на удаление сообщения

Вспомните, что удаление не имеет функции обратного вызова. В случае удаления запрос отправляется на сервер и сервер, в свою очередь, посылает запрос в базу данных на удаление соответствующей записи. Ниже приведен пример XML-запроса на удаление.

```
<Request>
   <TransactionType>DeleteMessage</TransactionType>
   <Data>
      <Message ID="5" />
   </Data>
</Request>
```

База данных

При работе с почтовыми услугами в базе данных в основном затрагиваются только таблицы Folders, Messages и Attachments.

Вначале сервер извлекает данные о папках из таблицы Folders в соответствии с идентификатором пользователя и отправляет их клиенту. Затем, когда пользователь нажимает на конкретную папку, сервер извлекает данные из таблицы Messages в соответствии с идентификатором папки и посылает клиенту список сообщений.

Когда пользователь нажимает на сообщение (для его просмотра), сервер извлекает данные из папки Messages (по идентификатору сообщения). При необходимости также запрашивается таблица Attacments (если в данном сообщении есть какие-нибудь приложенные файлы).

Запрос на удаление приводит к удалению сообщения из таблицы Messages. Также удаляются соответствующие приложенные файлы, если они есть.

Соединяя все вместе

К данному моменту мы рассмотрели три основные услуги Peachmail: регистрацию и вход под именем пользователя, адресную книгу и почтовые услуги. В этом разделе рассматриваются некоторые более сложные идеи ActionScript, использованные в Peachmail, включая специализированные классы и локальные соединения. Вдобавок мы также перечислим основные составляющие Peachmail-архитектуры, специализированные классы, созданные для Peachmail, и их назначение.

Продвинутый ActionScript

Одним из преимуществ Flash MX является использование компонентов для облегчения написания программ. Peachmail активно использует Macromedia-компонент списка (listbox). С учетом этого мы рассмотрим использованные Peachmail методы этого компонента. Мы также рассмотрим, как в Peachmail применяются объекты Local Connection (локальных соединений) для обмена данными между двумя внешними (external) swf-файлами. В конце мы рассмотрим создание прототипов (prototyping) и специализированных классов, переходя прямо к обсуждению Peachmail-архитектуры и соответствующих специализированных классов.

Компоненты Macromedia: применение компонента списка (listbox)

В Peachmail компонент списка применяется для показа адресов в адресной книге, списка папок сообщений и списка сообщений в папке. Мы обсудим наиболее широко используемые в Peachmail методы компонента списка: addItem(), getSelectedItem() setChangeHandler().

Функция addItem() вызывается при добавлении адреса в адресную книгу, при автоматическом обновлении списка папок (при входе пользователя в систему) или при обновлении списка сообщений (когда пользователь нажимает на конкретную папку). Например:

```
this.listbox.addItem(name, id);
```

создает в компоненте списка запись, содержащую имя и идентификатор, и отображает содержимое переменной name на экране - переменная name хранится в listbox как label, а переменная id- как data. К ним можно обратиться следующим образом:

```
mylabel = this.listbox.getSelectedItem().label;
mydata = this.listbox.getSelectedItem().data;
```

где getSelectedItem() возвращает выделенный в настоящий момент элемент списка. Обработчик события onChange() компонента списка вызывается автоматически, когда Джо выделяет элемент данного компонента. Например, в случае списка папок было показано, что с помощью вызова listbox метода setChangeHandler() событие onChange() устанавливается на получение сообщений выделенной папки. Приведенный ниже пример демонстрирует именно это ("onChange" здесь указывает на функцию onChange() класса folderList).

```
this.listbox.setChangeHandler("onChange", this);
    folderList.prototype.onChange = function () {
    // Вызываем callback-функцию, с выделенной папкой в качестве
    //параметра
    this.onSelect(this.listbox.getSelectedItem());
}
```

Последнее замечание в связи с компонентами списка в Peachmail относится к тому, как метод addItem() используется в классе emailList:

```
this.listBox.addItem(email.sender, email.subject, email.ReceiveDate, {id: email.id, email: email.email,
emailObject: email});
```

Вы видите, что методу addItem() передается более двух параметров. Это связано с тем, что компонент messages является специализированной версией стандартного Macromedia компонента listbox, где метод addItem() может вызываться с четырмя параметрами. Мы обсудим позже создание специализированных классов, а сейчас просто приведем код нового метода addItem().

```
FEmailListBoxClass.prototype.addItem = function(name, subject, date, data)
{
    if (!this.enable) return;
        this.dataProvider.addItem({name:name, subject:subject, date:date, data:data});
}
```

Соединение SWF с SWF: класс Local Connection (локальное соединение)

Объекты Local Connection появились в Flash MX и используются в Peachmail для обмена данными, например между newAddress.swf в одном окне и AddressBook.swf в другом окне. Локальное соединение требует наличия .swf-отправителя и .swf-получателя (рис. 7.16).

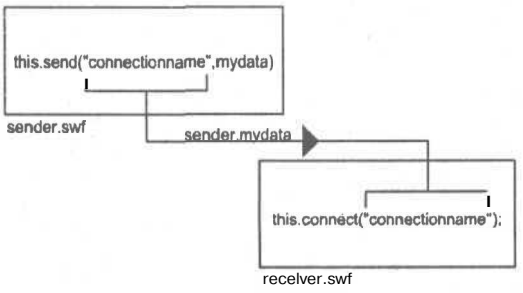

Рис. 7.16.
Соединяя два внешних .swf-файла

Файл-получатель (.swf) содержит следующий код:

```
RecvLC = new LocalConnection();
RecvLC.myfunc = function(msg) {
    trace("The sender has just sent me a message:");
    trace(msg);
}
RecvLC.connect("TheLC");
```

Для установления соединения с получателем файла-отправитель (.swf) содержит следующее:

```
SendrLC = new LocalConnection();
SendrLC.send("TheLC", "myfunc", "Your fly is open.");
```

Здесь получатель готовится к соединению с внешним источником, объявляя себя под именем "TheLC". Когда отправитель посылает "TheLC" -сообщение, он вызывает функцию получателя "myfunc", передавая в качестве параметра строку, которая выводится получателем (с помощью trace) в окне Flash MX Output.

В Peachmail локальное соединение с окном Address Book выглядит следующим образом:

```
newAddressConnection = new popupConnection ("newAddress", false);
newAddressConnection.newAddress = function (name, email) {
    // Создаем новый адрес
    addAddress(name, email);
}
```

Обратите внимание, что newAddressConnection является объектом класса popupConnection, а не LocalConnection. Класс popupConnection — это специализированный класс. Он создает объект LocalConnection и вызывает метод connect() для задания данного swf-файла в качестве получателя. Давайте взглянем на конструктор этого класса.

```
global.popupConnection = function(connectionName, popup) {
    // Сохраняем имя соединения
    this.connectionName = connectionName;
    // Сохраняем, всплывающее это окно или нет
    this.popup = popup;
    // Инициализируем очередь
    this.queue = [];
    // Начинаем popup соединение
    this.start();
};
// Наследуем от локального соединения
popupConnection.prototype = new LocalConnection();
```

Здесь класс popupConnection наследует от класса LocalConnection(), т. е. он наследует все атрибуты и методы класса LocalConnection(), а также может расширять (extend) свой родительский класс. Под расширением (extending) понимается воз-

можность добавлять новые методы, при этом сохраняя доступ к методам родительского класса, - вы увидите позже соответствующий пример. В последней строчке конструктора popupConnection() выполняется метод класса start():

```
popupConnection.prototype.start = function() {
    // Проверяем, popup это или нет
    if (this.popup) {
        // Соединяемся
        this.connect(this.connectionName+"Popup");
    } else {
        // Соединяемся
        this.connect(this.connectionName);
    }
    // Вызываем в другом окне функцию onConnect
    this.send("onConnect");
};
```

Так как в данном случае this.popup является ложным значением (заданным при создании newAddressConnection()), метод start() выполняет метод connect() класса popupConnection, по наследованию являющийся методом connect() класса LocalConnection():

```
this.connect("newAddress");
```

Так что теперь окно Address Book готово получать данные от любого .swf файла, (посылающего файлу newAddress.swf).

Перед тем, как перейти к коду на стороне получателя (окно newAddress), вам, наверно, любопытно узнать, почему в последней строчке метода start() вызывается метод send() класса popupConnection. Этот метод на самом деле является расширенным методом. При этом метод send() класса LocalConnection() остается таким же, но при вызове send() в объекте popupConnection на самом деле выполняется метод popupConnection.send():

```
popupConnection.prototype.send = function() {
    // Проверяем, popup это или нет
    if (this.popup) {
        // Добавляем к аргументам имя соединения
        arguments.unshift(this.connectionName);
    } else {
        // Добавляем к аргументам имя соединения
        arguments.unshift(this.connectionName+"Popup");
    }
    // Если это соединение или popup
    if (this.connected or this.popup) {
        // вызываем функцию в другом окне
        super.send.apply(this, arguments);
    } else {
        // Помещаем в очередь
        this.queue.push(arguments);
} };
```

Для доступа к родительскому методу LocalConnection.send() из объекта popupConnection можно воспользоваться вызовом super.send() и методом Function.apply():

```
super.send.apply(this, arguments);
```

В результате вызывается метод send() родительского класса с аргументами из списка arguments, а "this" используется для привязки родительского метода send() к классу popupConnection.

При вызове popupConnection.send() в файле AddressBook.swf в начало списка аргументов добавляется "newAddressGroup":

```
arguments.unshift(this.connectionName+"Popup");
```

а затем список аргументов добавляется в очередь:

```
this.queue.push(arguments);
```

так что в этот момент newAddressConnection.queue будет содержать:

```
[ ["newAddressPopup", "onConnect"] ]
```

Переходим к объекту LocalConnection на стороне отправителя, NewAddress.swf содержит следующий код:

```
newAddressConnection = new popupConnection("newAddress", true);
```

Рассматривая создание объекта newAddressConnection, мы видим, что одним из первых действий является задание самого себя в качестве получателя "newAddressPopup":

```
this.connect(this.connectionName+"Popup");
```

На следующем шаге вызывается метод send():

```
this.send("onConnect");
```

в котором в начало списка аргументов добавляется строка "newAdddress":

```
arguments.unshift(this.connectionName);
```

а затем вызывается Local Connection-метод send():

```
super.send.apply(this, arguments);
```

так что конечный вызов в файле newAddress.swf выглядит следующим образом:

```
newAddress.super.send("newAddress", "onConnect");
```

Здесь newAddress.swf соединяется с локальным соединением под названием "newAddress" и выполняет метод onConnect() данного локального соединения. Так как ранее AddressBook был задан получателем newAddress, то выполняется метод onConnect() объекта локального соединения AddressBook:

```
// Вызывается при соединении с popup
popupConnection.prototype.onConnect = function() {
    // Задаем флаг соединения
    this.connected = true;
    // Цикл по всем сохраненным спискам аргументов
```

```
    for (i = 0; i < this.queue.length; i++) {
    // Функция send
    super.send.apply(this, this.queue[i]); }
// Вызываем обработчик события onFinishQueue
this.onFinishQueue(); };
```

Как вы видите, флагу AddressBook под именем connected присваивается истинное значение, а затем AddressBook отвечает файлу newAddress.swf, проходя в цикле по очереди queue и применяя каждый элемент очереди к методу super.send() для получения:

```
newAddressConnection.super.send("newAddressPopup", "onConnect");
```

На стороне newAddress.swf вызывается соответствующий метод onConnect() и флагу connected файла newAddress.swf также присваивается истинное значение. Так как здесь очередь пуста, процесс соединения заканчивается. В результате всего этого мы получаем двустороннее соединение между AddressBook.swf и newAddress.swf.

Все, что обсуждалось выше, требовалось только для установления соединения между двумя .swf. Код, передающий новый адрес из newAddress.swf Address-Book.swf, выполняется при нажатии кнопки Create в newAddress.swf:

```
createHandler = function () {
    // Посылаем адрес
    newAddressConnection.send("newAddress", name.text, email.text);
    // Закрываем окно
    getURL("javascript:void(window.close());");
}
```

Это приводит к вызову метода newAddressConnection.newAddress(), описанного в начале данного раздела, который обрабатывает name.text и email.text данные из newAddress.swf.

Прототипы: Создание специализированных классов

По определению классом является набор атрибутов, методов и событий, используемых при создании объектов в среде Flash. Класс MovieClip(), например, содержит атрибуты: _x, _y, _width, _name и т. д.; методы gotoAndStop(), attach-Movie(), getDepth(), hitTest(), ... и события типа onEnterFrame and onLoad.

Peachmail содержит несколько специализированных классов, таких как ServerData(), buildBaseRequest(), folderList(), emailList(), и AddressBook(). Мы используем класс ServerData() в качестве примера создания специализированного класса.

В Peachmail server создается объектом класса ServerData() с помощью оператора new:

```
global.server = new ServerDataQ;
```

В результате server получает все атрибуты, методы и события класса ServerData(). Например, в следующих строчках:

```
server.setMethod(server.SEND_AND_LOAD);
server.setLanguage(server.JAVA);
```

методы setMethod и setLanguage принадлежат к классу ServerData().

Создание класса начинается с написания функции конструктора. Она вызывается при создании на основе класса нового объекта, например, с помощью оператора new, как в предыдущем примере. Целью конструктора является инициализация нового объекта заданными параметрами. Если инициализации не требуется, конструктором может быть пустая функция типа.

```
function myClass() { }
```

Ниже приведен конструктор класса ServerData():

```
function ServerData() {
    // Создаем константы языков
    this.NOT_SPECIFIED = 0;
    this.COLD_FUSION = 1;
    this.ASP = 2;
    this.ASPNET = 3;
    this.PHP = 4;
    this.JAVA = 5;
    // Создаем константы методов
    this.SEND = 1;
    this.LOAD = 2;
    this.SEND_AND_LOAD = 3;
    // Задаем язык по умолчанию
    this.setLanguage(this.NOT_SPECIFIED);
    // Задаем метод по умолчанию
    this.setMethod(this.SEND_AND_LOAD);
    // Задаем игнорирование белого цвета
    this.setIgnoreWhite(true);
}
```

Обратите внимание на ссылку this - она используется указания на атрибут и/или метод, принадлежащий данному классу. После создания конструктора создаются методы класса.

Методы класса определяются с помощью атрибута prototype. Например:

```
myClass.prototype.mymethod = function() {
// Код метода mymethod
}
```

Ниже приведены примеры некоторых методов класса ServerData(), а именно setLanguage() и execute().

```
ServerData.prototype.setLanguage = function(language) {
this.language = language;
}
ServerData.prototype.execute = function() {
    // Обнуляем статус
    this.setStatus("");
```

```
// Выбираем метод для выполнения
switch(this.getMethod()) {
// Обрабатываем отправку
case this.SEND:
    // Выполняем метод send
    results = this.executeSend();
    break;
// Обрабатываем загрузку
case this.LOAD:
    // Выполняем метод load
    results = this.executeLoad();
    break;
// Обрабатываем отправку и загрузку
case this.SEND_AND_LOAD:
    // Выполняем методы send и load
    results = this.executeSendAndLoad();
    break;
// Обрабатываем все остальное
default:
    // Метод не указан, создаем ошибку
    this.setStatus("Error: Invalid method defined!");
    return false
} // Конец switch
// Если есть результат, то ошибок не было
if(results) {
this.setStatus("Command executed properly");
}
// Возвращаем результаты выполнения метода
return results;
}
```

В конце давайте обсудим идею наследования. Класс можно сделать дочерним классом некоего родительского класса, наследуя таким образом все атрибуты, методы и события родительского класса. Ранее, в разделе о локальных соединениях, было показано, что класс popupConnection() наследует от класса LocalConnection() с помощью оператора new:

```
popupConnection.prototype = new LocalConnection();
```

Следовательно, popupConnection() является дочерним классом класса LocalConnection() и имеет доступ ко всем методам LocalConnection(). Затем, когда был определен метод popupConnection.send(), он заменил метод send() родительского класса (при этом остался доступ к методу родительского класса). Это и есть идея расширения класса: используя в качестве основы родительский класс и добавляя новые методы.

Архитектура Peachmail

В этом разделе еще раз затрагиваются некоторые идеи, заложенные в основу Peachmail-архитектуры, а именно: последовательность действий create-send-callback (создать, отправить, вызвать callback), стандартизированные транзакции и компонент списка (listbox). Каждой идее посвящен свой специализированный класс. Далее мы вкратце рассмотрим эти три идеи и соответствующие им классы.

Create-Send-Callback (класс ServerData)

Эта идея очень широко применяется в Peachmail. Класс ServerData() создан специально для того, чтобы помочь в создании XML-запроса транзакции, отправке этого запроса на сервер и, на основе ответа сервера, вызове callback-функции.

Константы класса ServerData(), применяемые при установке/получении (getter/setter) языка (language):

• NOT_SPECIFIED (default) • COLD_FUSION • ASP	• ASPNET • PHP • JAVA

Константы класса ServerData(), применяемые при установке/получении (getter/setter) метода (method):

• SEND • LOAD	• SEND_AND_LOAD (default)

В таблице 7.2 перечислены методы класса ServerData().

Таблица 7.2. Методы класса ServerData()

Метод	Описание
execute()	Выполняет метод, указываемый функцией getMethod(), с документом XML, заданным при помощи getDocOut() и/или getDocIn()
executeSend()	Вызывается методом execute() для выполнения XML Send с данными из getDocOut(); перед отправкой данные отформатированы в соответствии со спецификациями getLanguage()
executeLoad()	Вызывается методом execute() для выполнения XML Load с данными из getDocIn()
executeSendAndLoad()	Вызывается методом execute() для выполнения XML Send and Load с данными из getDocOut() и getDocIn() соответственно; перед отправкой данные отформатированы в соответствии со спецификациями getLanguage()
loaded (success)	Вызывается внутри класса по завершении операции XML Load, добавлен с целью возможного расширения класса в будущем; при этом также выполняется метод onLoad() класса ServerData
onLoad (success)	Метод-заглушка события onLoad. Для получения данных по окончании загрузки замените этот метод своим собственным
getMethod()	Методом по умолчанию является SEND_AND_LOAD

Метод	Описание
setMethod(method)	Используется с константами method (перечислены выше)
getDocIn()	Функция-получатель (getter) переменной Serverdata.docIn, содержащей xml-объект для ответа с сервера
setDocIn(xmldoc)	Функция для задания (setter) переменной Serverdata.docIn
getDocOut()	Функция-получатель (getter) переменной Serverdata.docOut, содержащей xml- объект для отправки на сервер
setDocOut(xmldoc)	Функция для задания (setter) переменной Serverdata.docOut
getURL()	Функция-получатель (getter) атрибута URL-объекта XML
setURL(url)	Функция для задания (setter) атрибута URL-объекта XML
getLanguage()	Значение по умолчанию NOT_SPECIFIED
setLanguage(language)	Используется с константами language (перечислены выше)
getIgnoreWhite()	Значение по умолчанию True
setIgnoreWhite(value)	Задает атрибут ignoreWhite документа XML
getStatus()	Функция-получатель (getter) переменной ServerData.status; Status содержит текстовое сообщение о результате транзакции
setStatus(status)	Функция для задания (setter) переменной ServerData.status
toString()	Возвращает строку "ServerData"

Стандартизированные транзакции (класс buildBaseRequest)

Все XML-транзакции в Peachmail похожи по формату. Общий стандарт XML-запросов и ответов во всех транзакциях очень сильно помогает при написании программы и облегчает ее улучшение в будущем, так как заранее заданный синтаксис позволяет быстро понять схему расположения данных. Обратившись к XML-запросам, рассмотренным в этой главе, можно видеть, что все они соответствуют следующему образцу:

```
<Request>
  <TransactionType>
    [ какой-то тип транзакции ]
  </TransactionType>
  <Data>
    < [data var 1] >
    [некоторые данные]
    </[data var 1]>
    .
    .
    .
    < [data var n] >
    [некоторые данные]
    </[data var n]>
  </Data>
</Request>
```

Класс buildBaseRequest() создан для того, чтобы собрать в одном месте весь код, связанный с созданием запроса, с целью облегчения возможных изменений в будущем. Большая часть инициализации запроса происходит в конструкторе класса, приведенном ниже.

```
function buildBaseRequest(transactionType) {
    // Создаем XML
    this.doc = new XML();
    // Создаем узел запроса
    var requestNode = this.doc.createElement("Request");
    // Создаем узел транзакции
    var transactionNode = this.doc.createElement("TransactionType");
    // Создаем значение транзакции
    var transactionNodeValue = his.doc.createTextNode(transactionType);
    // Инициализируем значение узла транзакции
    transactionNode.appendChild(transactionNodeValue);
    // Создаем узел данных
    var dataNode = this.doc.createElement("Data");
    // Добавляем узел транзакции к узлу запроса
    requestNode.appendChild(transactionNode);
    // Добавляем узел данных к узлу запроса
    requestNode.appendChild(dataNode);
    // Добавляем узел запроса к родительскому документу
    this.doc.appendChild(requestNode);
    // Возвращаем законченный документ
    return this.doc;
} // Конец функции buildBaseRequest
```

Ниже приведена табл. 7.3, перечисляющая методы класса buildBaseRequest().

Таблица 7.3. Методы класса buildBaseRequest

Метод	Описание
addData(newData, value, attributes)	Добавляет данные к XML
getDoc()	Возвращает XML документ

Обработчики событий, связанных с компонентами списка (listbox): классы AddressBook, folderList и emailList

Для работы с компонентами списка в Peachmail были созданы три класса: AddressBook(), folderList() и emailList(). Детали их использования уже были рассмотрены ранее. В качестве справочной информации ниже приведены конструкторы и таблицы с методами каждого класса (табл. 7.4–7.6):

Конструктор класса AddressBook():

```
_global.addressBook = function (listbox) {
    // Сохраняем ссылку на listbox
    this.listbox = listbox;
```

```
// Создаем массив для хранения адресов
this.addresses = [];
// Создаем объект для хранения ids
this.ids = {}; }
```

Таблица 7.4. Методы класса addessBook()

Метод	Описание
addAddress (name, email, id)	Добавляет адрес в список addressBook
deleteAddress (id)	Удаляет адрес из списка по данному id
deleteSelectedAddress()	Удаляет из списка выделенный в данный момент адрес
getSelectedAddress ()	Возвращает выделенный в данный момент адрес в списке
getSelectedIndex ()	Возвращает индекс выделенного в данный момент адреса в списке

Конструктор класса folderList():

```
global.folderList = function (listbox) {
    / Сохраняем listbox
    this.listbox = listbox;
    // Задаем обработчик onChange
    this.listbox.setChangeHandler("onChange", this);
    // Создаем массив папок
    this.folders = [];
}
```

Таблица 7.5. Методы класса folderList()

Метод	Описание
addFolder(name, id)	Добавляет папку в список Folders
deleteFolder(name)	Удаляет папку из списка по имени
deleteSelectedFolder()	Удаляет из списка выделенную папку
onChange()	Вызывает callback функцию

Конструктор класса emailList():

```
emailList = function (listbox) {
    // Сохраняем ссылку на listbox
    this.listBox = listbox;
    // Создаем новый массив для хранения сообщений
    this.emails = [];
    // Создаем объект для хранения ids
    this.ids = {};
    // Изменяем обработчик onChange
    this.listBox.setChangeHandler("onChange", this);
}
```

Таблица 7.6. Методы класса emailList()

Метод	Описание
setFolder(folder)	Задает текущую папку
addEmail(email, folder)	Добавляет сообщение в список Messages
addAttachment(emailId, name, id)	Добавляет приложенный файл в комбинированное окно Attachments
deleteSelectedEmail()	Удаляет из списка выделенное сообщение
getEmail(id)	Возвращает сообщение из списка по данному id
setAttachments(emailed, attachments)	Связывает сообщение с набором приложенных файлов
onChange()	Вызывает callback функцию
getSelectedEmail()	Возвращает выделенное сообщение

Заключение

В этой главе был по шагам рассмотрен процесс написания почтового клиента. Сначала технические требования клиента, далее написание прототипа программы, определение форматов данных, а затем написание кода. Было показано, как программа конструируется из клиента (Flash), сервера (Java) и базы данных (Microsoft Access). Также было продемонстрировано использование среды Flash MX.

Для обмена данными с сервером в Peachmail применяется набор действий create-send-callback. Программа использует в качестве формата данных XML и опирается на специальный класс для реализации точных определений данных. Также показано, как создаются прототипы специализированных классов в соответствии е определенными требованиями, например для обработки запросов или для работы с компонентом списка (listbox). Было продемонстрировано, как использовать локальные соединения для обмена данными между внешними .swfфайлами.

Но лучше всего то, что Джо, постоянный пользователь нашей программы, благодаря Peachmail сейчас наслаждается своим отпуском. Какой, однако, жизнерадостный парень, этот Джо!

8. Многоканальное приложение

Автор Вильям Б. Сандерс (William B. Sanders)

Новая технология, такая, как коммуникационный сервер Flash (Flash Communication Server MX, FlashCom), может быть интересна сама по себе, но это не главная причина ее создания. Коммуникационный сервер FlashCom был написан для решения определенных задач и, как вы вскоре увидите, может использоваться с большим практическим эффектом. В этой главе рассматривается практически применимая программа, совмещающая видео-, голос- и текст-чат, а также предоставляющая данные из базы данных. Совмещение разных каналов связи дает людям с разными уровнями доступа в Интернет возможность обмена информацией в реальном времени. В тестировании принимали участие пользователи с обычными модемами, DSL, кабельными модемами и T1/T3-линиями. После тестирования в программу были добавлены средства для изменения профиля Интернет-соединения (в зависимости от пропускной способности).

Первоначально программа с двусторонним видео-, голос- и текст-чатом рассматривалась в рамках центра службы поддержки покупателей (Customer Service). Поэтому она была написана на основе соответствующих требований (какая информация может потребоваться покупателю от представителя службы поддержки). Программа разбита на два модуля (покупатель и служба поддержки), при этом многие детали реализации в обоих модулях одинаковы. Оба модуля находятся в одной и той же папке приложений (application folder) и работают с общим сценарием на сервере.

Два видео окна и окно текст-чата занимают весь экран, не оставляя места для базы данных. Рассмотрев возможность открытия нового окна или перехода в другой кадр, я в конце концов решил наложить окно базы данных поверх окна текст-чата и задать средства управления таким образом, что база данных может быть активирована одним или двумя пользователями. База данных может быть размещена поверх текст-чата с помощью загрузки модуля базы данных на отдельный уровень. Такое наложение позволяет избежать проверки (можно ли отображать видео) каждый раз при появлении на странице нового видео.

Для того, чтобы лучше понять работу программы, мы сначала рассмотрим аппаратную и программную часть, необходимые для работы видео канала. Было проведено несколько тестов между Коннектикут и Лондоном с помощью Айо Бинити (aYo Binitie), бета-тестера и хозяина домена www.d-street.com. При тестировании с другим бета-тестером, Судхир Кумаром (Sudhir Kumar) из Индии я обнаружил, что мы могли поддерживать голос- и текст-чат даже при отсутствии видео камеры и телефона.

Я постарался сделать текст-чат программу по возможности простой. Она обеспечивает чат в одной комнате. Вначале написанная в расчете на двух человек программа была расширена на любое число участников. При общении с покупателем может потребоваться помощь других людей, а текст-чат не требует от соединения

большой пропускной способности (в отличие от видео- и голос-чата, которые ограничены в программе двумя людьми). В конце обсуждается общий интерфейс (generic front end) модуля по работе с данными (этот модуль может служить интерфейсом к любому числу частных модулей).

Настройка встроенного видео (Embedded Video)

Начать проект довольно просто. Для программы двустороннего видео- аудио-общения требуется только два видео окна. Вначале нужно настроить эти окна и дать им имена экземпляров (instance names). Соответствующие шаги:

1. Откройте панель Library. В среде Flash MX объект Embedded Video находится в всплывающем меню панели Library.
2. Для доступа к меню Options щелкните по стрелке всплывающего меню в верхнем правом углу панели Library. Это очень маленькая стрелка, которую можно легко не заметить в первый раз.
3. Из меню Options выберите New Video. Сразу после этого в нижней части панели Library должно появиться Embedded Video 1.
4. Перетащите один экземпляр Embedded Video 1 в верх левой половины рабочего поля, а потом разместите второй экземпляр под первым. Разместите верхний видео объект в месте с координатами X=12.5, Y=21.8, а нижний в месте с X=12.5, Y=182.9 (большие X на рис. 8.1). На слое под каждым видео-окном добавлены фоновые прямоугольники. Видео объекты при выделении отмечены синей кромкой, а цвет больших X совпадает со цветом слоя. Постарайтесь не поместить видео-объекты под непрозрачный объект: в этом случае они станут невидимыми, также, как и любой другой объект на рабочем поле.
5. Выберите верхний объект Embedded Video и в панели Properties дайте ему имя экземпляра rep.
6. Выберите нижний объект Embedded Video и в панели Properties дайте ему имя экземпляра cus.

На этом первая часть завершена. Оба объекта размещены на рабочем поле и им даны соответствующие имена экземпляров (как и в случае любого другого объекта Flash MX). Объекты Embedded Video можно динамически изменять и перетаскивать, так же как и клипы. (Когда программа поддерживает несколько динамически подключающихся и отключающихся пользователей, перетаскивание видео объектов очень полезно для реорганизации их на экране.)

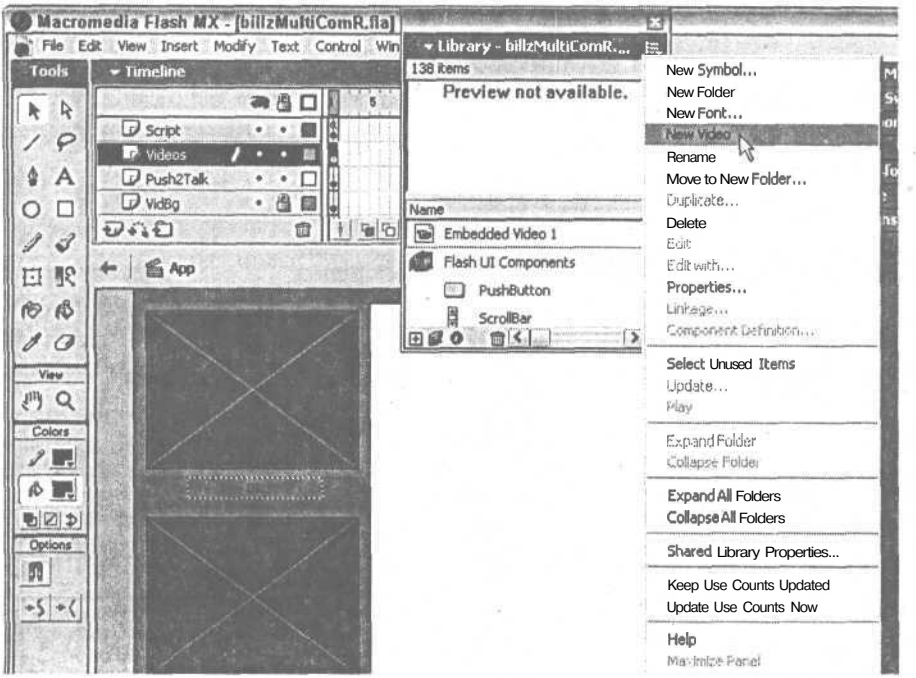

Рис. 8.1 . Размещение на рабочем поле двух экземпляров Embedded Video

Видео

Для видео чата нужно рассмотреть две вещи - аппаратные средства и программное обеспечение. К аппаратным средствам относятся видео камера и порт для ее включения в компьютер (например, порт USB, IEEE 1394). Под программным обеспечением понимаются настройки» которые можно изменить с помощью Communication ActionScript на стороне клиента. Хотя видео настройки также можно менять на стороне сервера (с помощью того же Communication ActionScript), в нашей программе видео контроль осуществляется только со стороны клиента.

Видео-камеры

Сконфигурировать аппаратную часть для коммуникационного сервера Flash MX оказалось одновременно просто и сложно. С камерами все было довольно просто. Я протестировал три Webcam и две цифровые видео камеры (DV), и все они работали нормально. Были использованы цифровые видео камеры Canon Ultura и Canon OpturalOO, в основном под Windows XP с картой Dazzle IEEE 1394. На компьютере Макинтош (400 МГц, 640 Мбайт RAM) я тестировал камеру iBOT и один из Firewire (IEEE 1394) портов. Также тестировалась камера D-Link USB WebCam под Windows XP и Макинтош. У меня вначале были проблемы с камерой PYRO как под XP, так и под Макинтош. Как я обнаружил позже, проблемы были связаны не с самой камерой. Камеры iBOT и PYRO не могут быть сконфигурированы с картой Dazzle IEEE 1394. Эта карта сконфигурирована таким образом, что она работает с DV-камерами, но не с WebCam, по крайней мере не с теми, которые

я пробовал. Я обнаружил, что PYRO работает нормально с XP Pro и общей (generic) картой ШЕЕ 1394. На табл. 8.1 показаны видео-камеры и Macromedia драйверы, протестированные и сертифицированные для использования с Flash Player 6. Три из тестированных мной камер (PYRO IEEE 1394, Canon Ultura и Canon Optura 100) не входят в этот список, но также работают нормально после корректной конфигурации.

Таблица 8.1. Видео камеры, драйверы, платформы и требования Flash Player 6

Производитель/модель	Версия драйвера	Платформы	Удовлетворяет требованиям
3com Home Connect	6.6.4	Win2000, MinMe, Win98	Yes
Creative WebCam Go Plus	1.0	MinMe, WinXP, Win98	Yes
D-Link USB WebCam (DBS-C100)	0921	Win2000, MinMe, Win98, WinXP, Mac 9.1, Mac 9.2	Yes
IBM Ultraport Camera 2	Most Current	Win2000	Yes
Intel Deluxe PC Camera	1.0	Win2000, WinXP	Yes
Intel Easy PC Camera	1.0	Win2000, Win98, WinXP, MinMe	Yes
Irez Kritter USB	1.02b4	Mac 9.1, Win2000, MinMe, Win98	Yes
Logitech QuickCam Traveler	6.0.1	Win2000	Yes
Logitech QuickCam Web	6.0.1	Win2000	Yes
Logitech QuickCam Express	6.0.1	Win2000	Yes
Logitech QuickCam Pro 3000	6.0.1	Win2000, Mac 9.1, MinMe, Mac 9.2, WinXP	Yes
Logitech QuickCam VC	Windows - 4.1.5 Macintosh - 2.1.2	Win2000, Mac 9.0, Mac 9.1, Mac 9.2, Mac OSX	Yes
Orange Micro iBOT Pro Firewire	4.10.22	Win2000, Mac OSX, Mac 9.2	Yes
Veo Stingray	5.0.0.0	Win98, WinXP, Win2000, MinMe	Yes

- Использование драйвера QuickCam Express версии 5 при определенных настройках может привести к отказу IE 5.5 под Win 2000. Для устранения проблемы обновите драйвер до версии 6.
- Использование камеры Kritter USB 2.1 (с прозрачным корпусом) иногда приводит к отказам в Flash Player 6 под Mac OS 5.9 и Windows.
- Если у вас несколько камер под операционной системой Windows: использование драйвера D-Link USB WebCam (DSB-C100) может привести к некорректной работе драйверов других камер. Для устранения проблемы установите еще раз все драйверы камер за исключением D-Link USB WebCam.
- Под Mac OS 9.2: камера D-Link USB WebCam (DSB-C100) каждый раз при использовании с Flash Player 6 демонстрирует очень медленное появление изображения (fade-in).

- Использование видео драйвера ATI Multimedia с Flash Player 6 может привести к отказам.
- Использование DV камер иногда приводит к проблемам: замедлению в работе и нарушения звука.
- Использование двух или более камер 3Com Home Connect требует очень много памяти.

Нужно также упомянуть проблемы, возникающие при использовании коммуникационного сервера Flash MX под Макинтош OS X. Во время написания книги основной Webcam-драйвер в OS X был довольно нестабилен. К моменту прочтения вами книги эта проблема, скорее всего, уже решена. Однако я обнаружил, что проблемы возникают только под OS X, под OS 9.2 все нормально, так что вы можете использовать коммуникационный сервер Flash MX под OS 9.2.

Примечание. *По непонятым причинам, когда моя цифровая видео камера (DV) была подключена к Макинтош под OSX, при тестировании нашей программы камера изменила фокус (telephoto shot). Это еще одна загадка для авторов драйверов OSX.*

Объект Camera и его настройки

Объект Camera является основным средством Communication ActionScript для установки видео настроек на стороне клиента. Так как в данном случае программа работает с аппаратным обеспечением (hardware), создание объекта Camera отличается от других Flash-объектов. Тем не менее после создания и помещения в ваш сценарий объекта Camera он обладает такими же характеристиками, как и другие объекты Flash MX ActionScript. Подходящей мишенью для объекта Camera является экземпляр Embedded Video, перенесенный на рабочее поле с панели Library. После создания объекта Camera его можно ассоциировать с экземпляром Embedded Video или просто с исходящим видео потоком (video stream).

В случае компьютера с одной видео-камерой объект Camera можно создать с помощью вызова функции get() без параметров. Например,

```
myCamera = Camera.get();
```

создает объект под названием myCamera, который адресуется так же, как и любой другой пользовательский объект, созданный во Flash MX ActionScript.

После создания объекта Camera нужно задать его атрибуты. В нашей программе используется следующий код Communication ActionScript (на стороне клиента):

```
// Настройка камеры
    cam = Camera.get();
    cam.setMode(160,120,9);
    cam.setQuality(0,90);
    cam.setKeyFrameInterval(5);
// Тип соединения
_root.conType.text="Default";
```

Эти настройки рассчитаны на быстрое Интернет-соединение, что-то типа DSL-линии, кабельного модема или локальной сети (LAN). Мы рассмотрим отдельно каждый из использованных в коде методов. (В модуле Customer пользователь может выбрать один из трех профилей, отражающих скорость соединения, поэтому необходимо сообщение о том, что текущие настройки используются по умолчанию. Это сообщение отражается в динамическом текстовом поле. В этом же поле демонстрируется также информация о всех других выбранных пользователем настройках.)

Camera.setMode(ширина, высота, fps, [favorSize])

Метод Camera.setMode() обычно довольно прост для использования. Если в объекте Embedded Video оставлен размер по умолчанию, то можно использовать те же значения для ширины (160) и высоты (120). Эти размеры можно увеличить или уменьшить, в зависимости от требуемого размера видео изображения в программе. Эксперименты с разными размерами показывают, что (учитывая неизбежные ограничения на пропускную способность Интернета) параметры по умолчанию являются неплохим компромиссом между качеством передаваемого изображения и разрешением.

По умолчанию fps (frames per second, частота смены кадров) задается равным 15. Однако в случае медленного соединения, в особенности в случае телефонного модема, эта скорость слишком высока. В нашей программе частота смены кадров задается равной 9. Такая скорость является довольно быстрой для DSL, как раз соответствует кабельному модему, и при этом несколько меньше, чем можно добиться в локальной сети (LAN). (Рекомендуемые настройки для разных типов соединений будут перечислены позже, в разделе "Типичные настройки объекта Camera для разных типов соединений".)

Для достижения более подходящего компромисса между размером кадра и частотой смены кадров введен дополнительный параметр favorSize, которому может быть присвоено булево значение. По умолчанию он равен true - поддержание стабильного размера кадра предпочтительней чем, частота смены кадров. Это означает, что в случае конфликта между размером кадра и частотой смены кадров размер будет более-менее постоянен, а частота смены кадров может сильно меняться. Присвоение favorSize значения false приведет к противоположному результату: частота смены кадров поддерживается постоянной за счет размера.

Camera.setQuality(bandwidth, frameQuality)

Параметр bandwidth задает максимальную пропускную способность (байт в секунду), требуемую для исходящего видео потока. По умолчанию это значение равно 16384, значение 0 означает, что Flash MX может самостоятельно выбрать максимальную пропускную способность на основе значения frameQuality.

Параметр frameQuality (качество кадра) может принимать значения в интервале от О до 100, при этом 100 означает максимальное качество. Повышенное качество требует большей пропускной способности; для вычисления правильного

значения требуемой максимальной пропускной способности задайте значение bandwidth равное 0. В нашей программе используется setQuality(0,90), т. е. минимальное качество на уровне 90 независимо от требуемой пропускной способности. Flash MX производит видео поток на заданном уровне качества. В табл. 8.2 показаны значения, рекомендуемые Macromedia для разных типов соединений.

Таблица 8.2. Рекомендуемые значения параметров Camera.setQuality() для разных типов соединений.

Пропускная способность	Уровень качества	Код
Модем	Низкое качество изображения, высокое качество движения	`cam.setQuality(4000,0)`
	Высокое качество изображения, низкое качество движения	`cam.setQuality(0,65)`
DSL	Низкое качество изображения, высокое качество движения	`cam.setQuality(12000,0)`
	Высокое качество изображения, низкое качество движения	`cam.setQuality(0,90)`
LAN	Низкое качество изображения, высокое качество движения	`cam.setQuality(400000,0)`
	Высокое качество изображения, низкое качество движения	`cam.setQuality(0,100)`

Camera.setKeyFrameInterval(c)

Идея ключевых видео кадров похожа на идею построения промежуточных изображений (tweening) в клипе Flash MX. Значение параметра keyframeInterval определяет интервал между ключевыми видео кадрами. Значение по умолчанию равно 15, в этом случае каждый 15-й кадр является ключевым. Остальные кадры интерполируются алгоритмом видео сжатия Flash MX.

Однако идея ключевых видео кадров все-таки немного отличается от tweening. Алгоритм видео-сжатия Flash MX передает только то, что изменилось с момента последнего ключевого кадра. В случае отсутствия сильного движения (например, в случае видео камеры наблюдения в закрытом офисе) интервал между ключевыми кадрами может быть довольно велик. Однако в случае интенсивной анимации количество ключевых кадров должно быть увеличено (для улучшения качества изображения и анимации).

Наилучшее значение параметра keyframeInterval можно определить только тестированием. Увеличение интервала между ключевыми кадрами уменьшает требуемую пропускную способность. Однако малое число ключевых кадров может увеличить время, требуемое для достижения определенного кадра (при произвольном перемещении по видео), так как может потребоваться интерполяция большого количества кадров. Так как в программе используется довольно низкая частота смены кадров (9), мы выбрали более короткий интервал между

ключевыми кадрами (5), что обеспечивает около двух ключевых кадров в секунду. Эти настройки вполне достаточны для общения.

Типичные настройки объекта Camera для разных типов соединений

При написании этой программы было довольно легко добавить кнопки на стороне покупателя. Предполагая, что у представителя службы поддержки будет стандартное Интернет-соединение, я добавил возможность изменения профиля соединения только на стороне покупателя. Ниже приведены соответствующие настройки объекта Camera в случае телефонного модема (56k), DSL и локальной сети (LAN).

```
// 56k Модем
cam.setMode(160,120,2);
cam.setQuality(0,75);
cam.setKeyFrameInterval(5);
// DSL
cam.setMode(160,120,5);
cam.setQuality(0,85);
cam.setKeyFrameInterval(5);
// LAN
cam.setMode(160,120,15);
cam.setQuality(0,90);
cam.setKeyFrameInterval(10);
```

Между этими установками можно перейти динамически с помощью кнопок на основе соответствующего типа соединения на стороне покупателя. С другой стороны, если вы планируете похожую программу и заранее знаете возможные типы соединений, вы можете заранее поменять настройки.

Выбор камеры

В конце рассмотрим также вопрос выбора камеры. На одном из моих компьютеров есть драйверы для DV- и WebCam-камер. Вначале я установил и протестировал DV-камеру, и ее драйвер позже блокировал работу камеры iBOT WebCam. Однако после успешного запуска программы среда Flash 6 Player обеспечивает довольно полезное контекстное меню. Для доступа к широкому набору настроек нужно предпринять следующие шаги:

1. Сделайте щелчок правой кнопкой мыши (Control-click под Макинтош) по видео окну.
2. В появившемся меню выберите Settings.
3. В окне Settings выберите иконку Webcam.
4. В появившемся всплывающем меню выберите нужный вам тип камеры и драйвера. На рис. 8.2 показан выбор Microsoft DV камеры и драйвера VCR.

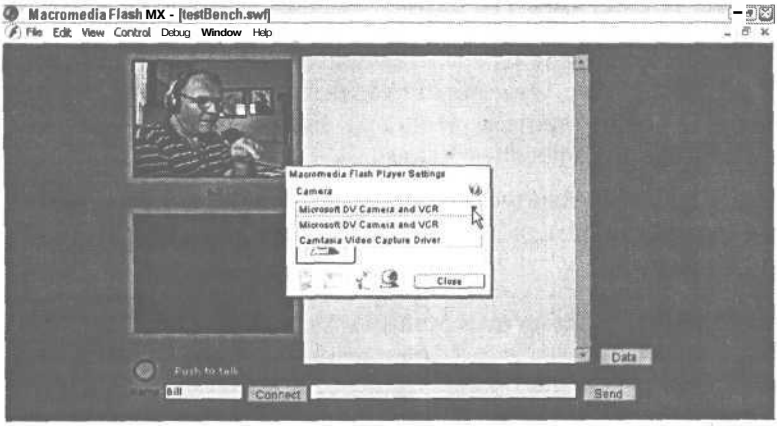

Рис. 8.2. Проверьте, что тип камеры и драйвера выбраны корректно

Если ваша камера все равно не работает, проверьте по табл. 8.1, совместимы ли ваша камера и драйвер с Flash 6 Player.

Микрофоны и настройка звука

Часть коммуникационного сервера Flash MX, связанная с аудио, довольно проста как в плане оборудования, так и в плане настроек. Во многих компьютерах, включая портативные (laptops) и все iMac, есть встроенный микрофон. В других есть разъем для внешнего микрофона, а в случае USB-микрофона в качестве разъема можно использовать USB-порт. Также довольно широк выбор гарнитур (наушники плюс микрофон).

Микрофон

В нашей программе очень легко настроить аудио. Мы тестировали программу с двумя USB- микрофонами разных типов под Windows XP, а также с USB микрофоном и встроенным микрофоном под Макинтош OS X. Все работало нормально, только с разными уровнями звука (при одинаковых настройках звука в нашей программе). Лучше всего работал Telex, USB-микрофон, протестированный под двумя разными платформами XP и под iMac с OS X. Вторым USB-микрофоном был Telex H-551 (гарнитура, наушники плюс микрофон). Гарнитура была гораздо, удобнее, чем наушники и микрофон по отдельности. В случае встроенного микрофона на iMac требовалось говорить довольно громко, даже со средним уровнем звука, в то время как для USB-микрофонов было вполне достаточно нормальной громкости речи. Судхир Кумар (Sudhir Kumar) из Индии, который также принимал участие в тестировании программы, использовал микрофон Intex, и качество получаемого звука было очень высоким, хотя были проблемы с остановкой звукового потока.

Объект Microphone и его настройки

В Communication ActionScript на стороне клиента основным средством тестирования аудио является объект Microphone. Этот объект создается практически так же, как и объект Camera, хотя в библиотеке (Library) Flash MX нет символа микрофона (аналогичного символу Embedded Video).

Для создания объекта Microphone используется метод get() без параметров, так же как и в случае объекта Camera. Например:

```
myMicrophone = Microphone.get();
```

Новый объект myMicrophone обладает всеми атрибутами и методами Microphone.

Код, задающий настройки микрофона в нашей программе, похож на соответствующий код в случае видео камеры, за одним важным исключением: в самом начале коэффициент усиления устанавливается равным 0:

```
// Задаем настройки микрофона
mic = Microphone.get();
mic.setGain(0);
mic.setRate(22);
```

В программе разговор возможен только при нажатии соответствующей кнопки и, когда кнопка не нажата, микрофон находится в отключенном состоянии (усиление равно 0).

Microphone.setGain(gain)

Метод setGain() позволяет задать коэффициент усиления микрофона. Нулевой коэффициент усиления вообще отключает микрофон до тех пор, пока пользователь не захочет что-нибудь сказать. Мы хотели вначале проверить, нельзя ли таким образом улучшить качество передачи звука (пользователи говорят по очереди), но особого выигрыша не получили. Главным преимуществом такого подхода оказалось подавление перекрестных помех.

Метод setGain() принимает значения в интервале 0–100. В нашей программе при нажатии кнопки разговора усиление задается равным 75. После отпускания кнопки усиление опять становится нулевым:

```
// Нажмите и говорите
talk.onPress = function() {
    mic.setGain(75);
};
talk.onRelease = function() {
    mic.setGain(0);
};
```

Объекту button дано имя экземпляра talk для того, чтобы данный код можно было поместить в том же месте, в котором находится и основной сценарий. В противном случае этот код нужно было бы ассоциировать с объектом кнопки. В соответ-

ствии с рекомендованными Macromedia методами написания сценариев весь код по возможности размещается в одном сценарии.

Вместо метода Microphone.gain() можно было бы также использовать атрибут muted объекта Microphone. Этот атрибут контролируется пользователем и доступен только для чтения. К сожалению, пользователь может контролировать этот атрибут только в момент соединения с камерой, когда Flash Player 6 обнаруживает объект Camera, объект Microphone и принимает или отвергает соединение с камерой и микрофоном. В результате атрибут Microphone.muted можно изменить только в момент установления и разрыва соединения с микрофоном.

Microphone.setRate(kHz)

Этот метод позволяет изменить частоту дискретизации микрофона (в килогерцах). Вместо интервала доступны только пять возможных значений: 5, 8 (значение по умолчанию), 11, 22 и 44. В программе задается частота 22. В модуле покупателя частота может измениться при изменении профиля Интернет-соединения. Строчка

```
mic.setRate(22);
```

в начале сценария устанавливает частоту дискретизации микрофона равной 22 кГц.

Типичные настройки объекта Microphone для разных профилей Интернет-соединения

Так же как и в случае объекта Camera, настройки объекта Microphone могут быть изменены покупателем для улучшения качества звука. Функции, изменяющие настройки камеры, также изменяют частоту дискретизации микрофона. Ниже приведены настройки микрофона в случае 5бк модема, DSL и LAN.

```
// 56к модем
mic.setRate(5);
// DSL
mic.setRate(11);
// LAN
mic.setRate(22);
```

Эти значения были выбраны в соответствии с использованными нами микрофонами. Вы можете поэкспериментировать с другими моделями и подобрать оптимальные значения в вашем конкретном случае (в зависимости от Интернет-соединения и типа микрофона).

Установка соединений

После создания объектов Camera и Microphone следующим шагом является установление соединений и связей между объектами. Вначале объект Embedded Video связывается с камерой, а затем связывается с исходящим или входящим потоком

видео. На первом шаге аппаратная часть и потоки связываются с объектом Embedded Video.

Video.attachVideo(source | null)

Объект Embedded Video создан с именем экземпляра Video. В нашей программе имена rep и cus соответствуют модулю службы поддержки и модулю покупателя (customer). Аргумент null означает остановку показа видео. Для показа видео в объекте Video не требуется коммуникационный сервер Flash MX. В программе используется только один метод объекта Video, attachVideo(), и не используется никаких атрибутов этого объекта.

Вначале нужно связать камеру с объектом Video (Embedded Video). Ниже приведен соответствующий код.

```
// Служба поддержки
rep.attachVideo(cam);
//Покупатель
cus.attachVideo(cam);
```

Здесь изображение с камеры перенаправляется в объект Video. После активации коммуникационного сервера Flash MX вы можете связывать ваше видео с исходящими видео потоками, а также связывать Video-объект со входящими видео потоками.

Связывания после установления соединения

В нашей программе требуются входящие и исходящие потоки аудио и видео. Сторона покупателя вначале посылает только то, что видит камера на стороне покупателя и что воспринимается микрофоном. На стороне службы поддержки принимаются аудио- и видео потоки от покупателя, а также посылаются соответствующие аудио- и видео потоки службы поддержки. Ниже приведена часть кода, посылающая и принимающая поток. Однако вначале должно быть установлено соединение:

```
//Соединяемся
function connect() {
    config.data.user = user;
    config.flush();
    // Соединяемся
    hookup = new NetConnection();
    hookup.connect("rtmp:/billzMultiCom", user);
    //
    // На стороне покупателя
    // Посылаем поток службе поддержки
    outStream = new NetStream(hookup);
    outStream.publish("cusStream", "live");
    outStream.attachVideo(cam);
    outStream.attachAudio(mic);
```

```
//
// На стороне службы поддержки
// Посылаем поток покупателю
inStream = new NetStream(hookup);
inStream.play("repStream");
rep.attachVideo(inStream);
```

Первые две строчки функции connect() относятся к инициализации текст-чата. Мы их пока пропустим, так как они не связаны с аудио/видеопотоками.

Объект NetConnection

Затем в функции connect() устанавливается соединение с приложением на коммуникационном сервере Flash MX, не с самим сервером. Переменной hookup присваивается созданный при этом объект (результат вызова new NetConnection()). Имя hookup (подключение) используется для обозначения того, что это объект соединения. После создания на стороне клиента объекта NetConnection вызываем метод connect() этого объекта (для идентификации приложения на коммуникационном сервере). Приложение идентифицируется не по имени файла, а по имени папки, в которой находятся файлы Flash MX SWF и вспомогательные файлы. С помощью протокола RTMP (Real-Time Messaging Protocol) сценарий передает идентификатор этой папки (в формате URI, Uniform

Resource Identifier). В соответствующей строке кода:

```
hookup.connect("rtmp:/billzMultiCom", user);
```

вы видите только одну косую черту (/). Одна косая черта может использоваться в том случае, когда приложение (SWF-файл) и сервер находятся на одной и той же машине (что чаще всего бывает в процессе разработки). Если сервер находится на другом компьютере, то нужно использовать две косые черты и тот же самый протокол RTMP, например:

```
hookup.connect("rtmp://someServer.someDomain.com/ЁbillzMultiCom", user);
```

При вызове метода connect() вдобавок к указанию идентификатора приложения также можно указать дополнительные параметры любого типа, которые будут переданы приложению. В нашем случае, параметр user относится к текст-чату и не связан с установкой аудио/видео соединения. (Однако он важен с точки зрения программы в целом, как вы увидите позже.)

Описанный выше процесс позволяет Flash-клиенту открыть TCP сокет с коммуникационным сервером для передачи RTMP-данных (аудио- и видео потоков). Через этот же сокет также может передаваться текст.

Передача исходящего потока с помощью NetStream(объект)

Объект NetConnection под названием hookup теперь нужно послать получателю. В нашей программе получателем является модуль покупателя или службы поддержки. Этот объект также должен принять входящие данные (приходящие через открытый сокет). Для этого нужно создать два объекта NetStream, один для входящих данных и один для исходящих.

Для отправки Flash-клиентом видео/аудио данных используется три NetStream метода:

```
NetStream.publish(whatToPublish | false [, howToPublish])
NetStream.attachVideo(source | null [, snapShotMilliseconds]))
NetStream .attachAudio(source)
```

Вначале создается объект NetStream для исходящих данных с соответствующим именем outStream:

```
outStream = new NetStream(hookup);
```

Затем нужно указать идентификатор (строку) для идентификации отправляемых (публикуемых) данных. Строка "cusStream" указывает на то, что это данные от покупателя. Это имя будет использовано подписчиками (на этот поток) для просмотра видео и прослушивания аудио данных. Вторым параметром функции publish() является одно из трех значений - record, append и live (запись, добавление и поток в реальном времени). Хотя по умолчанию используется "live", это значение тем не менее передается в явном виде, чтобы подчеркнуть, что это поток в реальном времени:

```
outStream.publish("cusStream", "live");
```

То, что публикуется объектом NetStream, может быть определено в терминах источников; в нашем случае источниками являются видео- камера и микрофон. Поэтому снова используется метод attachVideo(), но вместо связывания с Embedded Video сейчас связывание происходит с исходящим потоком. То же самое для аудио:

```
outStream.attachVideo(cam);
outStream.attachAudio(mic);
```

Это немного похоже на сплав леса по течению реки. Все, что вы бросаете в реку, может быть поймано ниже по течению.

Прием входящего потока с помощью NetStream(объект)

Прием входящего потока происходит аналогичным образом. Модуль покупателя посылает потоки с идентификатором "cusStream" и принимает "repStream". Сценарий создает объект NetStream, но вместо публикации вызывается метод play() (проигрывание потока). Новый объект NetStream соответственно называется inStream:

```
inStream = new NetStream(hookup);
inStream.play("repStream");
```

И наконец, входящий видео поток нужно где-то отобразить, поэтому снова вызывается метод attachVideo():

```
rep.attachVideo(inStream);
```

Входящий аудио поток автоматически направляется на динамик или наушники.

Текст-чат

Текст-чат добавлен на тот случай, если не работает аудио соединение. Иногда программа не может послать аудио поток из-за отсутствия микрофона или из-за проблем соединения. В этом разделе обсуждается весь код, относящийся к текст-чату. Для упрощения понимания текст-чата мы опустили части кода, относящиеся к объектам Camera и Microphone, а также к аудио- и видео потокам. Вначале рассматривается код на стороне клиента, а затем на стороне сервера.

Настройка текст-чата на стороне клиента

Функция connect() начинается с использования объекта SharedObject под названием config (весь код приведен ниже, в разделе "Текст-чат на стороне клиента"). Атрибут SharedObject.data доступен только для чтения; для присвоения значения нужно добавить тарибут и присвоить значение объекту, который вы хотите сделать общим (share). В нашем случае атрибуту user присваивается значение переменной user. Затем с помощью метода flash() локально перманентный (persistent) и общий (shared) объект записывается на диск.

Для проверки статуса соединения используется обработчик события onStatus того же самого объекта NetConnection под названием hookup. Затем экземпляру chatBox (большое динамическое текстовое поле) объекта hookup присваивается функция, помещающая в это поле текст из переменной msg. С помощью этой переменной выводятся сообщения от любого пользователя.

Далее переменная users определяется как SharedObject, для того чтобы сообщения от всех пользователей (напечатанное в их текстовом поле) попали в chatBox. Они соединяются, используя общий объект NetConnection (hookup) в качестве аргумента при вызове метода connect() объекта users (SharedObject). Сообщения последовательно добавляются в текстовое поле chatBox. Атрибут TextFiled.scroll задан равным 1000, чего должно быть вполне достаточно в обычном случае.

Далее функция sent() контролирует передвижения текста, введенного в текстовое поле message. Для этого используется метод NetConnection.call, с помощью которого вызывается метод или команда на стороне сервера. (См. раздел "Текст-чат на стороне сервера").

Текст-чат на стороне клиента

Оставшаяся часть кода текст-чата переключает кнопки интерфейса, создает объект config (SharedObject) и автоматически соединяет пользователя с чатом. При этом создается удаленный общий (shared) объект. Такие объекты создаются на стороне клиента, и при этом они также доступны серверу. Локальные общие (shared) объекты могут сохранять информацию на компьютере пользователя, но они недоступны серверу. Ниже приведен весь код текст-чата на стороне клиента.

```
function connect() {
    config.data.user = user;
    config.flush();
```

```
// Здесь NetConnection с именем объекта "hookup"
//.....
// Здесь Camera, Microphone и потоки
hookup.onStatus = function(info) {
        trace(info.code+newline);
        if (info.code == "NetConnection.Connect.Closed") {
            _root.chatBox.text = "";
        }
    };
    // NetConnection с основным текстовым полем
    hookup.chatBox = function(msg) {
    _root.chatBox.text = msg;
    };
    msg = "";
    // Создаем shared объект для всех пользователей
    users = SharedObject.getRemote("users", hookup.uri, false);
    users.onSync = function(list) {
    };
    users.connect(hookup);
    users.message = function(msg) {
        _root.chatBox.text += msg;
        _root.chatBox.scroll = 1000;
    };
    _root.sendButton.setEnabled(true);
}
// Посылаем сообщение из поля message в общую область чата
function send() {
    if (length(message.text)>0) {
        hookup.call("message", null, message.text);
    }
    message.text = "";
}
sendButton.setEnabled(false);
// Создаем объект config (SharedObject)
config = SharedObject.getLocal("config");
// Автоматическое соединение
user = config.data.user;
connect();
```

Контроль прокрутки текста

Одной из проблем, встретившихся при написании данной программы, была синхронизация действий, приходящих из Интернета и выполняемых локально (в сценарии). Как только текст достигает нижней части окна, необходима автоматическая прокрутка (чтобы пользователю не нужно было каждый раз прокручивать вручную). Часто трудно определить интервал времени между

началом (инициацией) действия и его выполнением. Проблемы синхронизации можно решить с помощью уникальной особенности Flash - помещения действия за пределы основной монтажной линейки и создания для этого действия отдельного клипа. Эта идея, предложенная Садхир Кумаром (Sudhir Kumar), сработала очень хорошо. Клип размещается на нижнем слое под встроенным видео (embedded video). Он состоит из двух кадров. В первом кадре содержится следующий код:

```
var lastPlace = _root.chatScroll.maxPos;
var pump = _root.chatScroll.getScrollPosition();
    if (pump != lastPlace) {
        _root.chatScroll.setScrollPosition(lastPlace);
    } else {
stop();
}
```

Максимальная позиция прокрутки (maxPos) сравнивается с текущей позицией (pump). Если они отличаются, метод setScrollPosition() вызывается до тех пор, пока они не станут равными. Затем клип останавливается до тех пор, пока функция send() не инициирует очередное действие play(). Имя (экземпляра) (pumper) данного клипа дожно быть включено в основной сценарий, для обеспечения доступа к клипу.

Клавиша Enter

Так как отправка сообщения более удобна при нажатии клавиши Enter (Return), чем при нажатии кнопки Send, нам требуется обработчик для этой клавиши. Функция send() переносит текст из поля message в общую область чата. Создается кнопка с названием экземпляра sender и ассоциируется со следующим кодом:

```
on (keyPress "<Enter>") {
    _root.send();
}
```

Эта кнопка находится под клипом на нижнем слое.

Текст-чат на стороне сервера

Сценарий на стороне сервера разделен на три части. В первой части инициализируются общие (shared) объекты для пользователей, соединяющихся с программой. Вторая (основная) часть соединяет пользователей с чатом, а третья посвящена окончанию соединения. Как и все сценарии, использующие коммуникационный сервер Flash MX и серверный Communication ActionScript, данный сценарий находится в файле mail.asc. В каждой папке, посвященной приложению Flash, может быть только один файл mail.asc.

В серверном Communication ActionScript самым популярным объектом является application. Этот объект связывает экземпляр вашего приложения с серверным сценарием, а также содержит информацию о данном экземпляре приложения.

Экземпляр объекта application может работать сразу с несколькими экземплярами приложения.

В первой части сценария (приведенного ниже) происходит начальное обнаружение приложения, использующего текст-чат, с помощью обработчика события application.onAppStart. Доступ к общему (shared) объекту осуществляется с помощью метода SharedObject.get. Этот метод возвращает ссылку на общий объект, указанный в первом аргументе ("users"). Затем он инициализирует экземпляр текст-чата (chatBox) и задает атрибут ID.

Во второй части находится функция, собирающая всех возможных участников. Это позволяет иметь сразу несколько участников в чате (больше двух). Хотя только двое могут использовать виде канал, количество участников в текст-чате не ограничено. Тестирование показало, что виде связь с неограниченным числом участников может очень сильно (экспоненциально в зависимости от числа участников) замедлить работу программы. Однако в случае текст-чата нет никаких проблем. Это может понадобиться, например, в том случае, когда на вопрос покупателя захотят ответить несколько представителей службы поддержки.

Последняя часть сценария просто удаляет пользователя из списка ID и обнуляет ID и имя. Ниже приведен весь описанный выше код.

```
// Весь серверный код на Rash Communication Server MX ActionScript
// Серверные сценарии сохраняются в файле main.asc
// Часть I–Обработчик события onAppStart (запуск приложения)
application.onAppStart = function()
{
trace("begin chat");
application.users = SharedObject.get("users", false);
application.chatBox = "";
application.nextId = 0;
}
// Часть II–обработчик события onConnect
application.onConnect = function(newClient, name)
{
newClient.name = name;
newClient.id = "u" + application.nextId++;
trace("connect: " + name + newClient.id);
application.users.setProperty(newClient.id, name);
application.acceptConnection(newClient);
// посылаем начальный chatBox
newClient.call("chatBox", null, application.chatBox);
// принимаем чат сообщение и отсылаем его обратно
newClient.message = function(msg) {
// Имя пользователя объединяется с сообщением
// и отсылается всем пользователям.
msg = this.name + ": " + msg + "\n";
application.chatBox += msg;
application.users.send("message", msg);
```

```
}
}
// Часть III-обработчик события onDisconnect .
application.onDisconnect = function(client)
{
trace("disconnect: " + client.id);
application.users.setProperty(client.id, null);
}
```

Помните о том, что этот код должен быть сохранен как файл main.asc и помещен в папку приложения. В папке приложения может быть только один серверный сценарий под названием main.asc.

Модуль службы поддержки

Так как этот модуль (показанный на рис. 8.3) скорее всего будет работать через достаточно быстрое Интернет-соединение, в нем отсутствуют кнопки, изменяющие профиль соединения. Вы также можете ввести свое имя в текстовом поле под верхним видео окном (так как предполагается, что у каждого представителя службы поддержки своя программа).

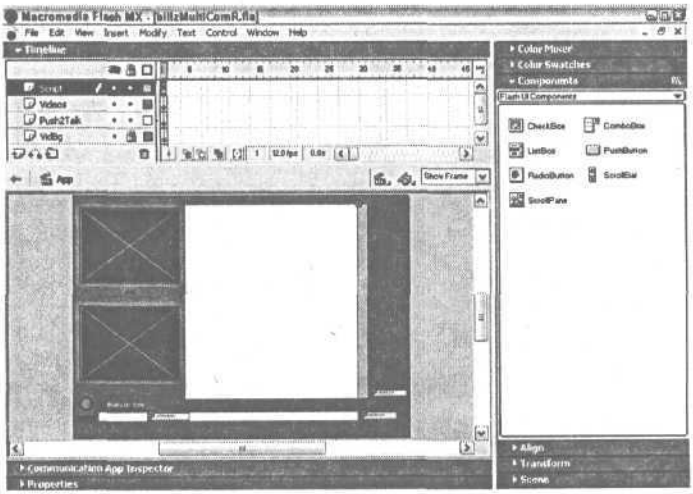

Рис. 8.3. Основной расклад и слои модуля службы поддержки

Организация объектов модуля службы поддержки

Объекты на рабочем поле в основном являются стандартными элементами Macromedia Flash MX. Нужно назвать и разместить на рабочем поле четыре стандартных компонента UI (интерфейса), пять текстовых полей, две кнопки

и клип. Все имена текстовых полей, кнопок и клипа являются именами экземпляров (instances). Ниже приведен полный списо.

Компоненты UI

Имя	Обработчик щелчка мыши
Connect	connect
Send	send
Data	dataCall

Имя	Мишень
chatScroll	chatBox

Текстовые поля

Имя экземпляра	Тип
repName	Dynamic
cusName	Dynamic
massage	Input

Текстовые поля

Имя переменной	Тип
user	Input

Кнопки

Имя экземпляра	Тип символа
talk	Button
sender	Button

Клип

Имя экземпляра	Тип символа
pumper	Movie Clip

Код на стороне клиента

Ниже приведен код модуля службы поддержки.

```
stop();
//Настройка камеры и микрофона
cam = Camera.get();
cam.setMode(160, 120, 5);
cam.setQuality(0, 90);
mic = Microphone.get();
mic.setRate(22);
mic.setGain(0);
rep.attachVideo(cam);
//Установление соединения
function connect() {
    config.data.user = user;
    config.flush();
    hookup = new NetConnection();
    hookup.connect("rtmp:/billzMultiCom", user);
    //Исходящий поток (покупателю)
    outStream = new NetStream(hookup);
    outStream.publish("repStream", "live");
    outStream.attachVideo(cam);
    outStream.attachAudio(mic);
```

```
    // Входящий поток (от покупателя)
    inStream = new NetStream(hookup);
    inStream.play("cusStream");
    cus.attachVideo(inStream);
    hookup.onStatus = function(info) {
        trace(info.code+newline);
        if (info.code == "NetConnection.Connect.Closed") {
            _root.chatBox.text = "";
        }
    };
    hookup.chatBox = function(msg) {
        root.chatBox.text = msg;
    }; "
    msg="";
    users = SharedObject.getRemote("users", hookup.uri, false);
    users.onSync = function(list) {
    };
    users.connect(hookup);
    users.message = function(msg) {
        _root.chatBox.text += msg;
        _root.chatBox.scroll = 1000;
    };
    root.sendButton.setEnabled(true);
}
function send() {
        if (length(message.text)>0) {
            hookup.call("message", null, message.text);
        }
        message.text = "";
        root.pumper.play();
}
sendButton.setEnabled(false);
// Отключаем до тех пор, пока не будет соединения
config = SharedObject.getLocal("config");
// Автоматически соединяемся
user = config.data.user;
connect();
talk.onPress = function() {
    mic.setGain(75);
};
talk.onRelease = function() {
    mic.setGain(0);
};
// Получаем данные
function dataCall() {
    if (fetchData.getLabel() == "Data") {
```

```
            loadMovieNum("data.swf", 2);
            fetchData.setLabel("Close");
        } else if (fetchData.getLabel() == "Close") {
            unloadMovieNum(2);
            fetchData.setLabel("Data");
        }
    }
```

Модуль покупателя

Модуль покупателя разрабатывался на основе модуля службы поддержки, с потоками, направленными в противоположную сторону. Он отличается от модуля службы поддержки в двух важных моментах. Во-первых, к нему были добавлены три дополнительные кнопки UI (для выбора профиля соединения) и дополнительное текстовое поле, сообщающее о текущем выборе профиля соединения (рис. 8.4). Это важно, так как на стороне пользователя пропускная способность Интернет-соединения может быть очень разной. Во-вторых, я начал работу над кодом, позволяющим авторизацию (модуль login), (рис. 8.5). Этот модуль сейчас просто является исходным пунктом для входа под именем пользователя. В данный момент он несколько избыточен, так как вход в систему происходит автоматически, тем не менее он полезен для демонстрации имени пользователя под видео окном.

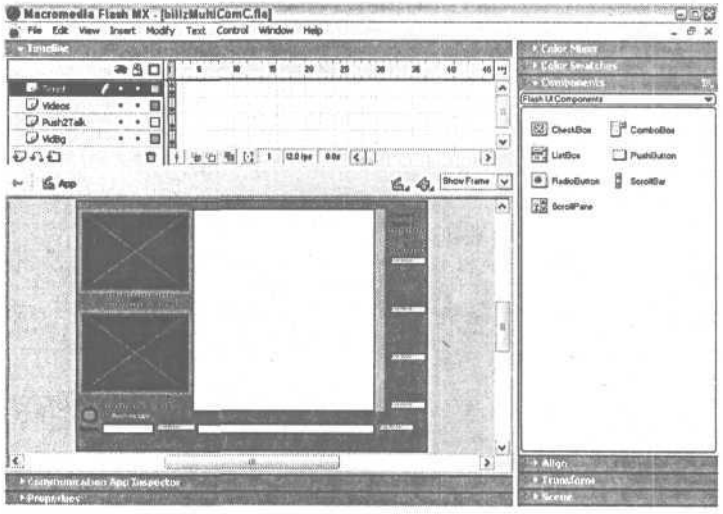

Рис. 8.4. Модуль покупателя содержит три дополнительные кнопки на правой стороне (для смены профиля соединения)

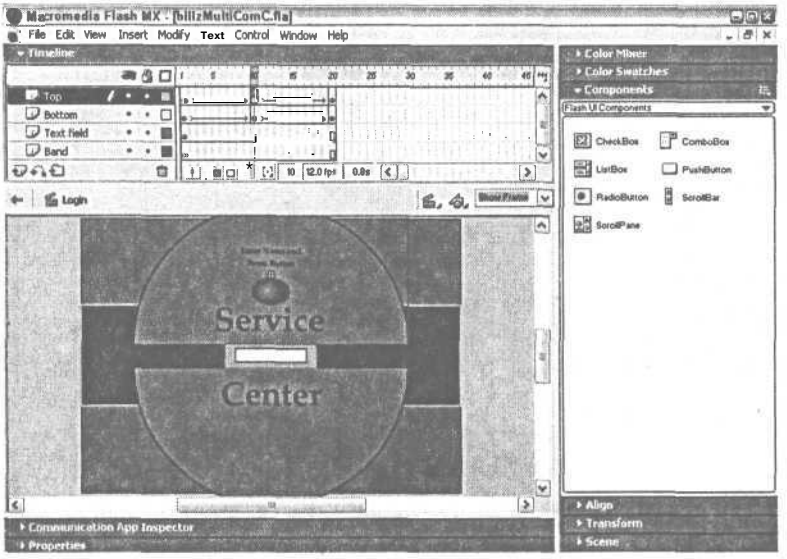

Рис. 8.5. Сцена Login позволяет пользователю ввести свое имя (которое потом появится под видео окнами)

Организация объектов модуля покупателя

Объекты на рабочем поле модуля покупателя в основном те же самые, что и в модуле службы поддержки; добавлены только три компонента UI, позволяющие изменить профиль Интернет-соединения. Ниже приведен полный список.

Компоненты UI

Имя	Обработчик щелчка мыши
Connect	connect
Send	send
Data	dataCall
LAN	lan
DSL	dsl
Modem	modem

Имя	Мишень
chatScroll	chatBox

Текстовые поля

Имя переменной	Тип
user	Input

Кнопки

Имя экземпляра	Тип символа
checkIn	Button
talk	Button
sender	Button

Текстовые поля

Имя экземпляра	Тип
repName	Dynamic
cusName	Dynamic
massage	Input

Сценарий на стороне клиента

Ниже приведен код модуля покупателя на стороне клиента.

Сцена: Login:

```
// Код кнопки checkIn
on(release) {
    _global.CusUser=_root.uname.text;
    _root.play();
}
```

Сцена: App. Кадр 1:

```
stop();
cusName.text = CusUser;
// Настройка микрофона и камеры
cam = Camera.get();
cam.setMode(160, 120, 5);
cam.setQuality(0, 90);
mic = Microphone.get();
mic.setRate(22);
mic.setGain(0);
cus.attachVideo(cam);
// Тип соединения
_root.conType.text - "Default";
// Установление соединения
function connect() {
    config.data.user = user;
    config.flush();
    hookup = new NetConnection();
    hookup.connect("rtmp:/billzMultiCom", user);
    // Исходящий поток (в службу поддержки)
    outStream = new NetStream(hookup);
    outStream.publish("cusStream", "live");
    outStream.attachVideo(cam);
    outStream.attachAudio(mic);
    //Stream in from customer rep
    inStream = new NetStream(hookup);
    inStream.play("repStream");
    rep.attachVideo(inStream);
    hookup.onStatus = function(info) {
        trace(info.code+newline);
        if (info.code == "NetConnection.Connect.Closed") {
            _root.chatBox.text = "";
        }
    };
    hookup.chatBox = function(msg) {
        root.chatBox.text = msg;
```

```
};
msg = "";
users = SharedObject.getRemote("users", hookup.uri, false);
users.onSync = function(list) {
};
users.connect(hookup);
users.message = function(msg) {
    root.chatBox.text += msg;
    root.chatBox.scroll = 1000;
};
root.sendButton.setEnabled(true);
}
function send() {
    if (length(message.text)>0) {
        hookup.call("message", null, message.text);
    }
    message.text = "";
    root.pumper.play();
}
sendButton.setEnabled(false);
// Отключаем до тех пор, пока не установим соединение
config = SharedObject.getLocal("config");
// Автоматическое соединение
user = config.data.user;
connect();
// Обработчик события onPress (разговор при нажатой кнопке)
talk.onPress = function() {
mic.setGain(75);
};
talk.onRelease = function() {
mic.setGain(0);
};
// Получаем данные
function dataCall() {
    if (fetchData.getLabel() == "Data") {
        loadMovieNum("data.swf", 2);
        fetchData.setLabel("Close");
    } else if (fetchData.getLabel() == "Close") {
        unloadMovieNum(2);
        fetchData.setLabel("Data");
    }
}
// Настройки
// Телефонный модем
function modem() {
    root.conType.text = "Phone Modem";
```

```
        cam.setMode(160, 120, 2);
        cam.setQuality(0, 75);
        cam.keyFrameInterval(3);
        mic.setRate(5);
    }
    // DSL
    function dsl() {
        root.conType.text = "DSL";
        cam.setMode(160, 120, 5);
        cam.setQuality(0, 85);
        cam.keyFrameInterval(5);
        mic.setRate(11);
    }
    // LAN–T1, T3, кабельный модем
        function lan() {
        root.conType.text = "LAN";
        cam.setMode(160, 120, 15);
        cam.setQuality(0, 90);
        cam.keyFrameInterval(10);
        mic.setRate(22);
}
```

Модуль данных: ActionScript, PHP и MySQL

Этот модуль написан только в качестве демонстрации. Демонстрационные данные находятся в базе данных MySQL, доступ к которой осуществляется с помощью сценария PHP. Этот модуль показывает возможность других способов обмена информацией (рис. 8.6). Несколько таких приложений могут быть наложены друг на друга в текст-чате. Те же самые функции могут быть обеспечены любой базой данных или серверной программой (backend middleware).

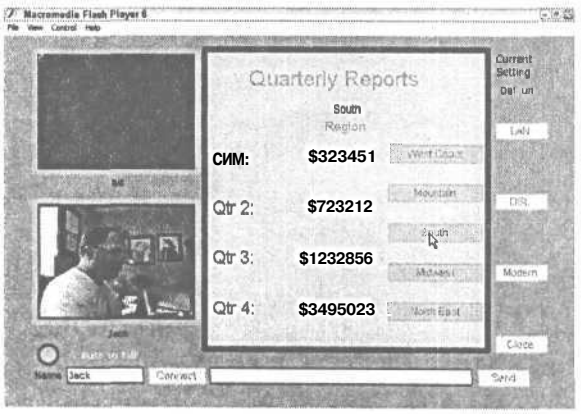

Рис. 8.6. Модуль данных обеспечивает дополнительный способ обмена информацией: данными из базы данных

Передача данных между Flash и PHP

На стороне клиента модуль состоит из пяти функций ActionScript. Похоже, что новый объект Flash MX LoadVars() является гораздо более эффективным для передачи переменных и данных между Flash cdjbv клиентом и сервером, чем старые функции loadVariables() и loadVariablesNum(). Метод sendAndLoad() позволяет вам отправить данные в одном объекте LoadVars() и получить в другом. Для каждой функции мы создаем LoadVars() объекты "Load" и "Hold". Объект "Load" загрузит (load) переменную для отправки сценарию PHP, а объект "Hold" сохранит MySQL данные, посланные через PHP. Затем, используя метод Load-Vars.onLoad и объект "Hold", данные заносятся в текстовые поля (с добавленным вначале символом $). Так как сценарию PHP передается переменная regNum, она была связана с объектом "Load". Со стороны PHP передаются переменные qt1wc, qt2wc, qt3wc, и qt4wc, соответствующие доходам в каждом из четырех кварталов года.

Организация объектов Модуля данных

На рабочем поле модуля данных находятся стандартные компоненты UI и текстовые поля Flash MX. Этот модуль является довольно общим примером (generic). В качестве последующего улучшения можно сделать все эти компоненты общими (shared) объектами, чтобы модуль мог быть открыт всеми пользователями, в данный момент соединенными с программой. Ниже приведен полный список компонентов:

Компоненты UI

Имя	Обработчик щелчка мыши
West Coat	west
Mountain	mountain
South	south

Имя	Мишень
North East	northeast

Текстовые поля

Имя экземпляра	Тип
q1	Dynamic
q2	Dynamic
q3	Dynamic
q4	Dynamic

Код ActionScript

```
function west() {
    root.region.text = "West Coast";
    westLoad = new LoadVars();
    westLoad.regNum = 0;
    westHold = new LoadVars();
    westHold.onLoad = function() {
        root.q1.text = "$"+this.qt1wc;
        root.q2.text = "$"+this.qt2wc;
```

```
        root.q3.text = "$"+this.qt3wc;
        root.q4.text = "$"+this.qt4wc;
    };
    westLoad.sendAndLoad("http://www.sandlight.com/getReport.php", westHold);
}
function mountain() {
    root.region.text = "Mountain";
    mtLoad = new LoadVars();
    mtLoad.regNum = 1;
    mtHold = new LoadVarsO;
    mtHold.onLoad = function() {
        root.q1 .text= "$"+this.qt1wc;
        root.q2.text = "$"+this.qt2wc;
        root.q3.text = "$"+this.qt3wc;
        root.q4.text = "$"+this.qt4wc;
    };
    mtLoad.sendAndLoad("http://www.sandlight.com/getReport.php", mtHold);
}
function south() {
    root.region .text= "South";
    southLoad = new LoadVarsO;
    southLoad.regNum = 2;
    southHold = new LoadVarsO;
    southHold.onLoad = function() {
        root.q1 .text= "$"+this.qt1wc;
        root.q2.text = "$"+this.qt2wc;
        root.q3.text = "$"+this.qt3wc;
        root.q4.text = "$"+this.qt4wc;
    };
    southLoad.sendAndLoad("http://www.sandlight.com/getReport.php", southHold);
}
function midwestO {
    root.region.text = "Midwest";
    mwestLoad = new LoadVarsO;
    mwestLoad.regNum = 3;
    mwestHold = new LoadVarsO;
    mwestHold.onLoad = function() {
        root.q1 .text= "$"+this.qt1wc;
        root.q2.text = "$"+this.qt2wc;
        root.q3.text = "$"+this.qt3wc;
        root.q4.text = "$"+this.qt4wc;
    };
    mwestLoad.sendAndLoad("http://www.sandlight.com/getReport.php", mwestHold);
}
function northeast() {
    root.region.text = "Northeast";
```

```
    northLoad = new LoadVars();
    northLoad.regNum = 4;
    northHold = new LoadVars();
    northHold.onLoad = function() {
        root.q1.text = "$"+this.qt1wc;
        root.q2.text = "$"+this.qt2wc;
        root.q3.text = "$"+this.qt3wc;
        root.q4.text = "$"+this.qt4wc;
    };
    northLoad.sendAndLoad("http://www.sandlight.com/getReport.php", northHold);
}
```

PHP-сценарий на стороне сервера

Мы постарались сделать сценарий PHP (приведенный ниже) очень простым (в пределах требований модуля данных). Из строки таблицы передается четыре значения, при этом номер строки передается из Flash MX. Переменная PHP под названием `$regNum` получает значения от 0 до 4, которые она использует для выделения полей MySQL, qt1, qt2, qt3 или qt4. (Для тех, кто незнаком с PHP, переменная, переданная в PHP, помещается в переменную с тем же именем плюс символ доллара [$] в начале.) Каждое поле каждого ряда содержит значения квартальных доходов маленькой компании.

Как только все данные помещены в переменные PHP, они форматируются для Flash MX. Формат:

```
varName1=val1&varName2=val2....
```

используется для помещения всех данных в одну переменную PHP, `$qtData`. С помощью PHP-оператора echo результат отправляется обратно в модуль данных:

```
<?php
// Настройки и соединение
$server="localhost";
// Имя сервера
$user="sandligh_streame";
// Имя пользователя
$pass="tincan";
// Пароль
$flashbase ="sandligh_flash1";
// Имя базы данных
$billz_table="reports";
// Имя таблицы
// Устанавливаем соединение
$hookup = mysql_connect($server, $user, $pass);
// Выбираем базу данных
mysql_select_db($flashbase,$hookup);
// Делаем query определенной таблицы базы данных
$result = mysql_query("SELECT * FROM $billz_table",$hookup);
```

```
$q1wc=(mysql_result($result,$regNum,"qt1"));
$q2wc=(mysql_result($result,$regNum,"qt2"));
$q3wc=(mysql_result($result,$regNum,"qt3"));
$q4wc=(mysql_result($result,$regNum,"qt4"));
// Форматируем для Flash
$qtData="qt1wc=$q1wc&qt2wc=$q2wc&qt3wc=$q3wc&qt4wc=$q4wc";
echo "$qtData";
?>
```

Следующие шаги

Одни приложения перестают изменяться в момент выхода в свет, а другие остаются развивающимися проектами. Для меня описанное выше приложение с несколькими способами обмена информацией остается развивающимся проектом. Каждый раз, когда мне казалось, что программа закончена, я обнаруживал, что ее можно улучшить, и мои друзья также показывали мне много возможных улучшений. Во время написания этой книги коммуникационный сервер (Flash Communication Server MX) только готовился к выходу в свет, и я уверен, что описанная выше программа - это только начало.

Я полагаю, что коммуникационный сервер MX фирмы Macromedia предлагает новые возможности для написания коммуникационных программ. С электронной почтой, миллионами веб-страничек и повсеместными мобильными телефонами можно только удивляться, что еще может что-то появиться в качестве средства общения. Кроме того, уже *в* течение многих лет можно (при наличии Webcam и Интернет-соединения) общаться через Интернет в видео/аудио режиме. Для меня и многих других, принимавших участие в написании коммуникационного сервера, идея создания и конфигурирования своей собственной коммуникационной программы оказалась очень привлекательной. В моем случае , я хотел написать программу с несколькими каналами общения, рассчитанную на двоих. Данный коммуникационный сервер является средством для решения любой проблемы коммуникации. В результате он не только удовлетворяет потребности многих людей в общении, но и позволяет экспериментировать с новыми способами общения.

На CD-ROM

В папке гл. 8 на CD-ROM-диска, приложенном к данной книге, вы найдете 10 файлов в папке под названием billzMuItiCom. В случае коммуникационного сервера Flash MX название папки играет большую роль. Папка является приложением (app). Имя папки должно соответствовать параметру функции NetConnection.connect() в модулях службы поддержки и покупателя:

```
hookup.connect("rtmp:/billzMultiCom", user);
```

Вы можете изменить название приложения на любое другое (например JoezMultiCom или SuezMultiCom). Однако при этом вам нужно также изменить строчку с вызовом функции NetConnection.connect() для сохранения соответствия с названием папки.

Из данных 10 файлов три файла FLA не обязательны для успешной работы приложения. (Если вы не пользуетесь браузером, HTML-файлы также не нужны.) Однако, если вы не планируете поместить приложение на удаленный сервер (remore hosting service), я советую вам оставить FLA-файлы в этой же папке. Это значительно облегчит внесение изменений в программу (меняете исходный код, а затем сохраняете и публикуете в том же месте).

Если вы пользуетесь пробной (trial) версией FlashCom или версией для разработчиков, вам нужно поместить папку с вашим приложением в папку под названием applications, которая находится в папке flashcom. Вы, наверно, захотите по возможности разместить папку flashcom в корне файловой иерархии вашего веб-сервера. Например, типичный путь под Windows XP Pro будет выглядеть:

```
C:\Inetpub\wwwroot\flashcom\applications\billzMultiCom
```

При запуске приложения один пользователь выбирает billzMultiComR.html (или .swf), а другой billzMultiComC.html (or swf). При использовании с браузером на вашем компьютере вы можете использовать адрес типа:

```
http://localhost/flashcom/applications/billzMultiCom/billzMultiComR.html
```

А человек, с которым вы хотите связаться, введет адрес:

```
http://IPaddress/flashcom/applications/billzMultiCom/billzMultiComC.html
```

где IPaddress является вашим IP-адресом (например, 12.201.34.701). Если вы захотите разместить ваше приложение на удаленном сервере, проверьте, стоит ли там коммуникационный сервер FlashCom и какие он предьявляет требования к конфигурированию приложений.

A. Многопользовательский сервер

Автор Михаэль Грюндвиг (Michael Grundvig)

При создании многопользовательских игр, таких, как в гл. 5, мы использовали сокет-сервер под названием ElectroServer и объект ActionScript под названием ElectroServerAS. В этом приложении мы обсудим, что такое сокет-сервер, как он работает и как его можно использовать в ваших собственных играх. Иначе говоря, мы детально рассмотрим ElectroServer. Также будут описаны детали установки ElectroServer на многих типах компьютеров, от домашнего компьютера до многопроцессорного сервера фирмы Sun.

Что такое сокет-сервер?

Перед тем как углубиться в детали установки и использования сокет-сервера, желательно вначале понять, что это такое. Вкратце, сокет-сервер (также называемый многопользовательским сервером) - это сервер, ожидающий (listen) входящих соединений от клиентов на заданный заранее порт и позволяющий этим клиентам обмениваться информацией друг с другом по стандартному протоколу (например XML в случае ElectroServer). Так как это довольно длинное определение, давайте разделим его на несколько частей. Мы начнем с обзора деталей, связанных с Интернетом.

Основы Интернета

Хотя я уверен, что каждый читатель этой книги очень часто пользуется Интернетом, тем не менее довольно многие разработчики не знакомы со всеми деталями работы Интернета. Я всегда считал, что, чем больше вы знаете о чем-нибудь, тем эффективнее вы можете этим пользоваться. В случае сокет-сервера дополнительная информация поможет вам избежать неожиданных проблем с установкой приложений и их безопасностью.

IP-адрес

Хотя IP (Интернет-протокол) - адрес находится далеко не на самом низком уровне в сети Интернет, это самый низкий уровень, который для нас важен. IP-адрес - это адрес компьютера в Интернете. Хотя есть и исключения, проще всего предполагать, что у каждого компьютера в Интернете свой уникальный IP-адрес (рис. АЛ). IP-адрес, вместе с некоторыми другими технологиями Интернета, позволяет любому компьютеру найти и установить соединение с любым другим компьютером в Интернете.

IP-адрес обычно выглядит как XX.XX.XX.XX, где каждый XX-элемент является числом из одной, двух или трех цифр. Например, адрес одного из веб-серверов сайта Macromedia.com - 65.57.83.12 .

Эти числа часто называются dotted quads. Один элемент IP-адреса называется октетом (octet), так как максимальный размер элемента 2^8, а значение элемента лежит в пределах 0-255. Вначале может показаться, что число возможных IP-адресов очень велико и их хватит надолго; однако на самом деле мы уже приближаемся к пределу. Хотя текущая спецификация IPv4 позволяет около 4.3 миллиарда адресов, довольно скоро этого может оказаться мало, так как в связи с бурным ростом Интернета IP-адресов каждый день требуется все больше.

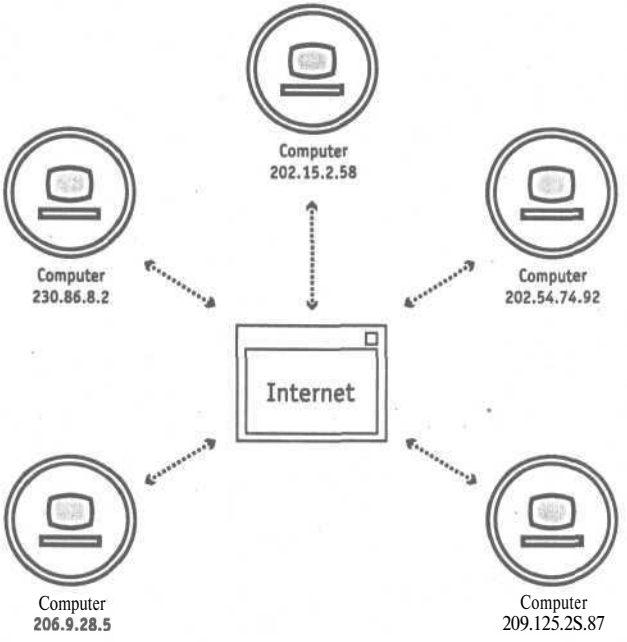

Рис. А.1. Как правило, у каждого компьютера в Интернете есть свой уникальный IP-адрес

Поэтому в данный момент разрабатывается новая спецификация Ipv6. Она добавляет возможность лучшей передачи мультимедийных потоков и повышает производительность. Но самое главное, эта спецификация увеличивает размер IP-адреса с 32 до 128 бит.

Как получить IP-адрес? В случае домашних пользователей, IP-адрес назначается при соединении с провайдером. Если вы пользуетесь телефонным модемом, то скорее всего при каждом выходе в Интернет вы получаете новый адрес. Кабельный и DSL-модемы обычно работают так же, но, так как в этом случае вы всегда соединены с Интернетом, IP-адрес чаще всего не меняется до тех пор, пока вы не перегрузите машину. Однако есть и исключения. Многие DSL-провайдеры используют технологию PPPoE (Point-to-Point Protocol over Ethernet), позволяющую отключиться от Интернета и подключиться заново без перегрузки машины. В этом случае ваш IP-адрес может измениться. Также можно "освободить" ваш текущий IP-адрес и получить другой, но этот метод зависит от вашей операционной системы и не гарантирует получение новогя IP-адреса.

На рис. А.2 показано типичное соединение домашнего компьютера с Интернетом через DSL или кабельный модем. У вас также может быть *статический IP-адрес*. В этом случае ваш IP-адрес остается тем же самым (постоянным) даже между перезагрузками машины. Сервера в Интернете почти всегда используют постоянный IP-адрес, и, за некоторыми исключениями, такой адрес почти всегда удобней динамического.

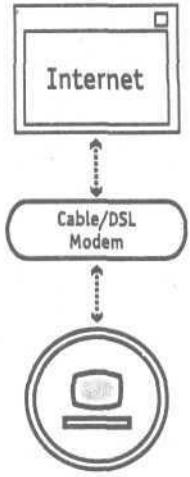

Рис. А.2. DSL или кабельный модем соединяет домашний компьютер с Интернетом

Нужно ли находиться в сети Интернет для того, чтобы получить IP-адрес? Совсем необязательно! Каждый компьютер, поддерживающий TCP/IP (используемый Интернетом сетевой протокол), всегда может использовать loopback IP-адрIP-адрес. Это зарезервированный IP-адрес, и вы можете соединиться ним, используя IP-адрес 127.0.0.1 или имя хоста localhost. Данный IP-адрес используется для тестирования и диагностики и может также применяться разработчиками Flash для локальной работы с сокет-сервером (изолированно от сети). Мы обсудим это более подробно позже.

Ранее было сказано, что у каждого компьютера в Интернете свой уникальный IP-адрес, за некоторыми исключениями. Сейчас мы поговорим об этих исключениях. Вы, наверно, слышали о средствах защиты сети (брандмауэр — firewall) и о маршрутизаторах (router). Эти устройства могут быть использоваты для изоляции локальных IP-адресов от Интернета. Я затронул эту тему потому, что она напрямую связана с задачей запуска сокет-сервера на вашем домашнем компьютере таким образом, чтобы он был доступен через Интернет.

Для иллюстрации приведем пример из реальной жизни. Если маленькая компания желает обеспечить своим 20 сотрудникам доступ к Интернету через единственное ISDL-соединение, как она может это сделать? Можно было бы связаться с провайдером и выделить 20 IP-адресов для своей сети. Однако этот способ решения задачи связан со значительными проблемами администрирования и поддержки. Более простым решением было бы использование технологии трансляции IP-адреса (NAT, network address translation). Эта технология позволяет одному компьютеру обеспечивать доступ в Интернет для всех остальных (служить в качестве шлюза). Этот компьютер будет иметь одно внешнее соединение (в Интернет) со своим внешним IP-адресом и одно внутреннее соединение с внутренним адресом (рис. А.3). Это делается обычно с помощью двук сетевых карт. Под внутренним адресом подразумевается IP-адрес, понятный только внутренней сети. Часто эти адреса имеют 10 в качестве первого октета. У всех остальных компьютеров во внутренней сети также свои уникальные внутренние IP-адреса, и они выходят в Интернет через шлюз.

Я затронул этот вопрос потому, что многие люди с кабельным модемом или DSL также пользуются маленькими маршрутизаторами/брандмауэрами (от таких компаний, как Linksys, 3Com или Intel). Когда вы пользуетесь таким устройством, то

ваш сокет-сервер (на компьютере с внутренним IP-адресом) не будет виден в Интернете (до тех пор, пока вы не измените конфигурацию вашего маршрутизатора/брандмауэра). А именно сокет-сервер запустится без всяких проблем, но, когда вы укажете ваш IP-адрес и порт, люди не смогут соединиться с вашим сокет-сервером через Интернет. В этом случае нужно использовать специальное свойство вашего устройства (DMZ, перенаправление порта или перенаправление IP-адреса) для того, чтобы позволить внешним клиентам соединяться с вашим сокет-сервером. DMZ (демилитаризованная зона) означает, что ваш компьютер (с сокет-сервером) является незащищенной частью вашей сети, и маршрутизатор/брандмауэр будет пропускать к нему все внешние соединения. Перенаправление порта или IP-адреса означает, что маршрутизатор/брандмауэр вместо блокирования запросов по данному порту или IP-адресу будет перенаправлять их на ваш компьютер. Еще одним решением было бы полное открытие соответствующего порта на вашем маршрутизаторе/брандмауэре. Детали по реализации всех этих решений можно посмотреть в документации вашего маршрутизатора/брандмауэра.

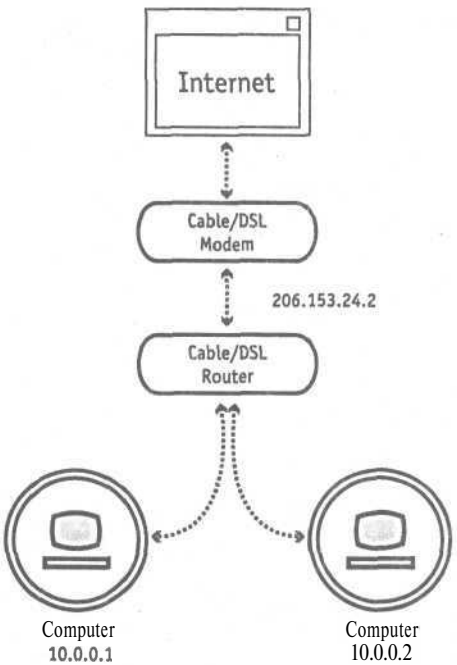

Рис. А.3. Компьютеры во внутренней сети с IP-адресами, понятными только в пределах внутренней сети

Порты

До этого мы говорили только об адресе компьютера в Интернете. Это довольно хорошая общая аналогия, но ее желательно детализировать. Типичный компьютер в Интернете имеет один IP-адрес, но при этом на нем может одновременно работать много разных Интернет-ориентированных приложений. Как вы знаете, веб-браузеры, программы мгновенного обмена сообщениями, почтовые клиенты, многопользовательские игры и другие программы могут работать одновременно независимо друг от друга.

При наличии только одного адреса как ваш компьютер понимает, какие (полученные) данные для какой программы? Для этого и используются порты. С каждым IP-адресом ассоциировано 65.536 портов. Если сравнить IP-адрес с адресом дома, то порт будет соответствовать номеру квартиры.

Это означает, что ваш компьютер в принципе может одновременно обмениваться информацией с 65.000+ удаленных компьютеров (в реальной жизни домашнему компьютеру это обычно не под силу, но некоторые сервера вполне способны с этим справиться). Так что при соединении с удаленным компьютером используется не только IP-адрес, но и конкретный порт, для уникальной идентификации как компьютера, *так и* приложения, с которым вы хотите обмениваться данными.

Вы, наверно, подумали, что 65,536 портов должно быть более чем достаточно. В принципе это так, но тем не менее нужно быть осторожным в выборе порта. Многие порты зарезервированы под определенные сервисы, такие, как HTTP, FTP, POP, SMTP, и многие другие. В общем, желательно никогда не использовать для ваших приложений порты со значением меньше чем 1024. В случае Flash и XMLSocket это выполняется автоматически - Flash не может использовать порт со значением меньше чем 1024. В табл. АЛ приведены несколько примеров зарезервированных портов.

Таблица A.1 Зарезервированные порты часто используемых приложений

INTERNET SERVICE	TCP PORT
HTTP	80
HTTPS	443
SMTP	25
POP3	110

INTERNET SERVICE	TCP PORT
FTP	21
Telnet	23
NMTP	119

Сокеты

Сокет - это просто один конец сетевого соединения. Каждый сокет задается с помощью IP-адреса и порта. При работе серверной программы она ожидает (listens) соединение от клиента на данный сокет. Клиент, например веб-браузер, знает IP-адрес (или имя хоста) и порт приложения и пытается соединиться. Если все проходит успешно, сервер принимает (accepts) соединение, а затем переносит это соединение на другой сокет (выбранный более-менее произвольно). Перенос происходит для того, чтобы сервер мог начать ожидать новое соединение по первому сокету (с целью одновременной работы сразу с несколькими клиентами). После переноса соединения на произвольно выбранный сокет клиент и сервер могут свободно начать обмен данными.

Давайте продолжим наш пример с веб-браузером. Вы запускаете ваш любимый браузер и набираете http://www.macromedia.com. Благодаря DNS (Domain Name System, связывает имена с IP-адресами) набранное имя превращается в IP-адрес 65.57.83.12. Так как вы используете протокол HTTP, браузер по умолчанию предполагает порт 80. Этой информации вполне достаточно для попытки установить

сокет-соединение с 65.57.83.12:80 (что браузер и делает). Веб-сервер сайта Macromedia принимает соединение и переносит его на своей стороне на другой сокет. В этот момент браузер уже может запрашивать веб-страничку. Как только получение странички закончено, браузер автоматически закрывает соединение. Это довольно общая картина того, что происходит.

Как это связано с Flash?

Мы обсудили основные принципы установления TCP-соединения между компьютерами в Интернете и как при этом ведет себя обычное приложение. Теперь как это происходит в Flash? Именно таким же образом!

Flash работает с сокетами точно так же, как и ваш Веб-браузер. Единственное отличие в протоколе. В веб-браузере используется HTTP-протокол. При получении электронной почты - POP3 или IMAP. При отправлении почты - SMTP. В случае Flash используется XML, определяемый вами в соответствии с вашим приложением. (И Flash не выбирает какой-либо определенный порт, так что вы можете выбрать любой порт (за исключением тех, которые меньше 1024).

Теперь, когда мы рассмотрели необходимую дополнительную информацию, давайте поговорим конкретно о сокет-серверах и о том, как они работают с Flash.

Сокет-сервер

Сокет-сервер - это просто сервер, ожидающий соединения от Flash клиентов по данному сокету и принимающий данные соединения. Это довольно простое определение, и вы, наверно, удивляетесь, чем отличается сокет-сервер от веб-сервера. Двумя вещами: сокет-сервер использует другой протокол и соединения являются state-full.

state-full-соединение - это соединение, сохраняющее информацию о клиенте в перерывах между обменами данных. Это значительно увеличивает возможности Flash по сравнению с приложениями, основанными на HTTP-протоколе. Протокол HTTP не сохраняет информации о клиенте. Есть много методов, добавляющих к HTTP возможность *кажущегося* сохранения информации, но на самом деле веб-страничка не знает, посетили ли вы ее пять минут или пять дней назад. В случае *state-less* приложений (таких, как использующих HTTP-протокол) приложение должно само определить, кто вы, а также сохранить все необходимые данные. А в случае сокет-сервера соединения остаются открытыми до тех пор, пока вы сами их не закроете (в коде или при закрытии Flash-приложения). Это означает, что через одно соединение может пройти большое число транзакций и что сервер будет помнить о клиенте между транзакциями. Данная особенность необходима для таких функций чата, как история, точный подсчет человек в комнате и т, д.

Протоколом веб-сервера является HTTP. Протоколом высокого уровня для сокет-сервера, как мы уже сказали, является XML. Именно в этом проявляются большие возможности сокет-сервера. Благодаря использованию XML в качестве протокола высокого уровня вы можете задать свой собственный (user-defined) протокол с учетом требований вашего приложения.

Введение в ElectroServer

Как видите, сокет-сервер является простым, но довольно мощным инструментом. Сокет-сервер можно реализовывать разными способами, но общая идея остается той же самой. Для Flash существует несколько главных сокет-серверов и довольно много второстепенных. Мы познакомим вас с самым известным нам сервером, лучше всего приспособленным для игровых приложений: ElectroServer. Это высокопроизводительный сокет-сервер, написанный компанией Electrotank (www.electrotank.com) в расчете на многопользовательские Flash-игры. В данной книге все многопользовательские игры применяли ElectroServer и различные его особенности. На нескольких следующих страницах мы рассмотрим эти особенности, а также как устанавливать, использовать и управлять (administer) сокет-сервером ElectroServer.

Примечание: *Если вы собираетесь сейчас запустить этот сервер, прочитайте сначала подраздел "Файл атрибутов" в разделе "Конфигурирование сокет-сервера ElectroServer", приведенном ниже.*

Примечание: *На приложенном CD-ROM-диске вы найдете полную версию ElectroServer и ключ демо-лицензии. Демо-лицензия ограничивает BectroServer пятью одновременными соединениями и работает для любого IP-адреса. Это означает, что с сервером могут связаться одновременно не более пяти человек. BectroServer в демонстрационной моде должен иметь возможность выхода в Интернет при каждом своем запуске, в противном случае он не запустится. Если вы хотите получить самую последнюю информацию о сокет-сервере ElectroServer, либо получить копию ElectroServer, позволяющую более пяти одновременных соединений, посетите сайт www.electrotank.com/ElectroServer .*

Особенности

Сокет-сервер ElectroServer обладает несколькими уникальными особенностями (отсутствующими в других сокет-серверах), позволяющими сделать игру более интересной. Давайте начнем с более простых, последовательно переходя к более сложным. Многие особенности могут быть сконфигурированы через файл атрибутов (который будет описан немного позже в разделе Конфигурирование сокет-сервера ElectroServer, подраздел Файл атрибутов).

Комнаты. ElectroServer поддерживает понятие комнат для всех чатов и игр. Это означает, что после соединения с сервером вы не сможете ничего сделать до тех пор, пока не войдете в какую-нибудь комнату.

Комнаты могут быть видимые и скрытые (в этом случае они не показаны в списке комнат). Об этом заботится объект ElectroServerAS (детально описанный в приложении В).

При изменении числа участников в комнате все остальные комнаты получают сообщение об изменении и о новом размере комнаты. Это удобно для отслеживания количества участников в любой данной комнате. Комнату можно также сконфигурировать так, что она будет игнорировать эти сообщения (для повышения производительности), и многие люди предпочитают такую конфигурацию.

Список комнат. Все клиенты (пользователи) получают точный список всех видимых комнат на сервере. Этот список также содержит количество участников для каждой видимой комнаты.

Список пользователей. Внутри каждой комнаты есть список пользователей данной комнаты. Этот лист всегда свежий, так как он обновляется всякий раз, когда кто-то входит или выходит.

Частные сообщения. ElectroServer поддерживает возможность отправки пользователями частных сообщений любому человеку на сервере.

Администраторы. ElectroServer поддерживает понятие администраторов. Администраторы - это пользователи, которые могут удалять участников и запрещать их повторное появление (рассматривается ниже). Для создания или удаления регистрационной записи (account) администратора нужно использовать встроенную программу - консоль администратора (объясняется позже).

Удаление пользователя (Kick user). Если пользователь нарушает правила поведения на сервере (и? поверьте мне, это случается *очень часто),* то вы можете удалить данного пользователя. Это отличается от похожей особенности в IRC (где пользователь удаляется только из комнаты). В ElectroServer команда kick полностью отключает пользователя от сервера и выдает ему сообщение о причине удаления. Эта команда доступна только пользователям с привилегиями администратора.

Запрещение пользователя (Ban user). Если пользователь ведет себя в особенности плохо, вы можете совсем запретить ему вход на сервер. Этот запрет основан на IP-адресе, так что его можно обойти. Тем не менее он постоянен - если вы запретите пользователя и перегрузите сервер, запрет все равно останется в силе.

Администраторы могут отменить запрет (удалить пользователя из списка запрещенных) с помощью специальной административной программы.

Ведение журнала (Logging). ElectroServer обладает расширенными возможностями для ведения журнала. При запуске сервера он инициализирует все необходимые компоненты и выводит (logs) информацию об этом на экран. После запуска всех компонентов и начала нормальной работы (ожидания соединений от пользователей) сервер начинает вести журнал в месте, указанном в файле атрибутов. По требованию сервер также может прокручивать (roll) файлы журнала при запуске (переименовывать старый журнал, сохраняя под другим именем, и начинать новый). Сервер поддерживает разные уровни сохранения информации, позволяя вам определить, насколько много (или мало) данных должно быть сохранено. Все сообщения об ошибках, удалении и запрете пользователей сохраняются в любом случае.

Языковой фильтр. ElectroServer поддерживает возможность фильтрации слов. Запрещенные слова перечислены в текстовом файле, имя которого указано в файле атрибутов. Данная особенность может быть включена и выключена, она также имеет много конфигурационных параметров, включая возможность удаления пользователей.

Системные сообщения. Системное сообщение передается всем пользователям во всех комнатах, независимо от их статуса. Они могут быть использованы для предупреждения об ожидаемых событиях, таких, как перезагрузка сервера или чат со знаменитостью. Системные сообщения могут посылать только администраторы.

Переменные, связанные с комнатой. Переменная, связанная с комнатой, задается пользователем на уровне комнаты. Эти переменные можно по-разному конфигурировать. Например, переменную можно сконфигурировать на удаление самой себя в тот момент, когда пользователь выходит из комнаты. Переменную можно блокировать от последующей перезаписи, или ее можно сделать постоянной (persistent), существующей даже тогда, когда ее автор покинул комнату. Эти переменные автоматически рассылаются всем пользователям в момент их создания, изменения или удаления. При входе в комнату пользователь также получает все переменные, связанные с комнатой. Эти переменные являются одной из особенностей ElectroServer, предназначенной именно для написания игр.

Сериализация объектов[1] (Object serialization). Эта важная особенность сокет-сервера ElectroServer на самом деле основана на возможностях объекта на стороне клиента. Объект ElectroServerAS поддерживает способность брать большинство ActionScript-объектов, не связанных с аппаратными ресурсами (такими, как видео и аудио клипы) и посылать их в комнату, в которой находится пользователь, или даже способность их употребления в качестве переменных комнаты. Это значительно облегчает пересылку сложных структур данных.

Отслеживание обновлений. ElectroServer поддерживает возможность автоматической проверки на предмет появившихся обновлений его кода через Интернет. При запуске сервера он соединяется по HTTP-протоколу с www.electrotank.com и проверяет доступные версии. Если предлагаются более новые версии, ElectroServer сообщает вам об этом.

Администрирование. ElectroServer поддерживает несколько административных (консольных) программ, облегчающих администрирование сервера. С их помощью можно управлять регистрационными записями администраторов и списком запрещенных пользователей.

Установка сокет-сервера ElectroServer

Хоть это и профессиональный сервер, его вполне можно запускать на вашей рабочей машине. На самом деле это рекомендованный способ написания многопользовательских программ.

Существует несколько способов установки ElectroServer. Так как ElectroServer написан полностью на Java, его можно запускать на любой платформе, поддерживающей виртуальную Java машину (JVM, Java Virtual Machine) версии 1.3.1. В данный момент компания Sun Microsystems предлагает JVM для Windows, So-

1. Сериализация - сохранение объектов в потоке. - *Примеч. науч. ред.*

laris и Linux; Java поддерживается также на многих других платформах, таких, как Mac OS X и HP-UX.

Независимо от платформы, вначале нужно установить JVM. Для Windows, Solaris и Linux ее можно найти на сайте Java.Sun.com (http://java.sun.com). В случае Mac OS X виртуальную машину можно получить как Mac OS X Java Runtime Environment (http://developer.apple.com/java).

Установите JVM в соответствии с инструкциями, приведенными на сайте (с которого вы взяли JVM). После окончания установки наберите в командной строке java; вы должны получить ответ типа:

```
C:\WINDOWS\Desktop>java
Usage: java [-options] class [args...]
(to execute a class)
or java -jar [-options] jarfile [args...]
(to execute a jar file)
```

(Это в случае JVM версии 1.4, в вашем случае ответ на команду java может выглядеть немного по-другому.) С этого момента ваш компьютер должен поддерживать Java-программы и вы можете перейти к установке непосредственно сокет-сервера ElectroServer, которая должна быть очень простой.

Установка под Windows

С CD-ROM-диска (приложенного к данной книге) запустите программу setup.exe (инсталлятор для Windows). Вы найдете ее в папке Demos\ElectroServer\Windows. После запуска этой программы установите ElectroServer в соответствии с появившимися инструкциями. В процессе установки создается новая группа в меню Start и сценарий для запуска и остановки сервера. Программа установки также создает программу удаления (uninstaller).

Установка под Unix

Найдите на CD-ROM диске папку Demos\ElectroServer\Unix. Скопируйте файлы из этой папки в то место, в котором вы хотите разместить ваш сервер. (Обычно это что-то типа /usr/local/ElectroServer.) Затем запустите сценарий Setup.sh. Этот сценарий извлекает содержимое .tar-файла в данную директорию. На этом установка сервера закончена.

Установка на других платформах

Найдите на CD-ROM-диске папку Demos\ElectroServer\Generic. В ней находится файл с расширением .zip. Поместите содержимое этого файла туда, где вы хотите установить сервер.

В отличие от других способов установки, в этом случае не создаются автоматически сценарии запуска и остановки сервера.

Конфигурирование ElectroServer

Теперь, когда JVM и сервер уже установлены, вам нужно их сконфигурировать, с учетом вашего окружения, требуемых особенностей и опций при запуске. Конфигурирование сокет-сервера ElectroServer очень просто: все, чем нужно управлять или что нужно изменить, доступно либо через простые консольные программы, либо через текстовый файл атрибутов.

Файл ***атрибутов***

Конфигурирование сокет-сервера ElectroServer в основном выполняется с помощью файла атрибутов ElectroServer.properties, который находится в той же директории, в которой находиться и установленный сервер. Вы можете открыть данный файл с помощью любого редактора, например Notepad или VI. Так как различные конфигурационные опции сервера документированы прямо в этом файле, мы не будем здесь их рассматривать. Любые изменения в файле атрибутов требуют последующего перезапуска сервера.

Для запуска сервера очень важны две опции. Перваяиз них

```
General.LicenseFileLocation
```

Она указывает на местонахождение файла с лицензией, без которой сервер не запустится. По умолчанию эта опция указывает на файл с демонстрационной лицензией, позволяющей запускать ElectroServer на любом IP-адресе, но с количеством одновременных соединений не более пяти. Эта лицензия также требует наличия доступа к Интернету, иначе сервер не запустится. Если какие-либо настройки в файле атрибутов не соответствуют вашему файлу лицензий, то сервер сообщит вам об этом и остановится.

Второй важной опцией является

```
ChatServer.IP
```

Она указывает IP-адрес чат-сервера. По умолчанию она равна 127.0.0.1. Это локальный адрес вашей машины (loopback IP), как мы обсуждали ранее. Вам нужно изменить этот адрес, если вы хотите, чтобы другие люди могли соединиться с вашим сервером через Интернет.

Регистрационные записи (accounts) администраторов

Если вы хотите иметь на вашем сервере администраторов, то вам нужно установить в файле атрибутов опцию ChatServer.AdministrationEnabled true. После этого нужно добавить администраторов с помощью административной программы сокет-сервера ElectroServer. Вы можете запустить эту административную программу следующим образом:

Windows. Либо выберите в меню Start опцию Start Administrator, либо перейдите в папку, в которой установлен ElectroServer, и запустите файл StartAdministrator.bat.

Unix. Запустите файл StartAdministrator.sh в папке с сокет-сервером ElectroServer.

Другие платформы. Перейдите в папку, в которой установлен ElectroServer, и запустите:

```
java -cp ElectroServerV2.jar com.electrotank.electroserver.admin.ElectroAdmin
```

Обратите внимание, что это должна быть одна команда.

Запустив данную административную программу, выберите опцию Manage Administrator Accounts и далее следуйте в соответствии с инструкциями.

Языковой фильтр

Языковой фильтр позволяет блокировать на чат-сервере любые слова из указанного вами списка. Вы просто указываете местонахождение файла со списком слов в файле атрибутов ElectroServer.properties. Этот список должен быть текстовым файлом в формате:

```
ПлохоеСлово1
ПлохоеСлово2
...
ПлохоеСлово10
```

Этот файл можно изменить в любое время, но для того, чтобы изменения вступили в силу, нужно перезапустить ElectroServer.

Запрещенные IP-адреса

Если какие-то IP-адреса были запрещены и вы хотите удалить их из списка запрещенных, вам нужно запустить административную программу сокет-сервера ElectroServer (таким же образом, как было описано выше в случае создания новых администраторов). После запуска выберите опцию Manage Banned IP Addresses и далее следуйте в соответствии с инструкциями.

Запуск/остановка сокет-сервера ElectroServer

Запуск/Остановка сокет-сервера ElectroServer так же просты, как и в случае описанной выше административной программы. Как вы понимаете, перед тем, как начать тестирование, чат или игру, сервер нужно запустить.

Запуск сервера

Windows. Выберите из меню Start опцию Start ElectroServer или перейдите в папку с установленным сервером и запустите файл StartElectroServer.bat.

Unix. Из папки с сервером запустите StartElectroServer.sh.

Другие платформы. Выполните команду

```
java -cp ElectroServerV2.jar
com.electrotank.electroserver.ElectroServer
ElectroServer.properties
```

из папки, в которой установлен сервер.

Остановка сервера

Windows. Выберите из меню Start опцию Stop ElectroServer или перейдите в папку с установленным сервером и запустите файл StopElectroServer.bat.

Unix. Из папки с сервером запустите StopElectroServer.sh.

Другие платформы Выполните команду

```
java -cp ElectroServerV2.jar
com.electrotank.electroserver.StopES
ElectroServer.properties
```

из папки, в которой установлен сервер.

Сокет-сервер ElectroServer появился почти вместе с созданием Flash 5 и за это время использовался во многих разных приложениях. Он был применен с большим успехом многочисленными компаниями на многих платформах. Приложив небольшие усилия, вы должны без проблем адаптировать ElectroServer к вашим требованиям.

B: Объект ElectroServerAS

Автор Джоб Макар (Jobe Makar)

Объект ElectroServerAS позволит вам легко создавать чаты, многопользовательские игры и любые другие многопользовательские приложения, которые вы захотите написать. Это объект ActionScript со многими атрибутами и методами, облегчающими написание программ с Macromedia Flash. С помощью этого объекта вы можете обмениваться информацией с сокет-сервером ElectroServer, не написав ни одной строчки в XML; ElectroServerAS сделает это за вас, если вы хотите послать чат-сообщение, вам нужно только написать строчку ActionScript кода типа:

```
ElectroServerAS.sendMessage(info, "room")
```

Здесь переменная info содержит сообщение, которое вы хотите отправить. Второй параметр указывает, хотите ли вы отправить сообщение всей комнате или только одному пользователю. При выполнении этой строчки кода объект ElectroServerAS определяет нужные XML-тэги, форматирует данные и посылает их на сервер.

Но объект ElectroServerAS может больше, чем просто отправлять и принимать сообщения. Он может создавать на сервере переменные, связанные с указанной комнатой. Для облегчения написания многопользовательских игр была также добавлена возможность отправки объектов другим пользователям (с помощью сценария WDDX_ms.as, написанного Бранденом Халлом (Branden Hall) из Fig Leaf Software). Это большое преимущество по сравнению с другими многопользовательскими серверными системами (сервер плюс клиент), так как в этом случае если игрок делает ход в игре, то вместо формирования сложной XML-строки с кучей атрибутов для передачи хода можно просто послать сам объект. Также очень полезна возможность создания на сервере переменных комнаты. В случае создания, изменения или удаления этих переменных всем участникам комнаты рассылается уведомление. В результате облегчается создание, например, карточной игры типа покера или пасьянса (в которых требуется уведомлять всех игроков о раскладе карт). Без этой особенности нужно было бы использовать сложную схему уведомлений через двусторонние сообщения, и эта схема мне очень не нравится. Переменные, связанные с комнатой, позволяют обойти эту проблему, а также предлагают много других новых возможностей в играх.

Примечание: *Вы, конечно, всегда можете создавать чаты и многопользовательские игры без применения объекта ElectroServerAS, но это потребует много работы - написания и синтаксического разбора XML-документов, а также многих часов, посвященных отладке программы.*

Я уже упоминал о большом преимуществе сокет-сервера ElectroServer — о возможности передачи объектов ActionScript другим пользователям. Я хочу подчеркнуть, что другие многопользовательские серверы в принципе также могут это делать, так как на самом деле это особенность объекта ElectroServerAS, а не сокет-

сервера. Если вы используете другой сокет-сервер, вы можете сами добавить эту возможность, но это потребует много времени.

Для использования большей части объекта ElectroServerAS вы должны понять сущность обработчиков событий. Существует много многопользовательских событий, таких, как получение чат-сообщения, хода в игре, обновление списка участников комнаты, получение вызова на игру от другого пользователя. Хотя вам скорее всего не понадобятся все методы и атрибуты объекта ElectroServerAS, вы наверняка хотите знать, что они предлагают и как их использовать. В этом приложении мы перечислим и опишем каждый метод и атрибут объекта ElectroServerAS.

Развитие продолжается!

Так же как и в случае всякого изобретения, облегчающего нашу жизнь, мы постоянно находим новые способы улучшения объекта ElectroServerAS. Для получения самой свежей версии этого объекта (а также исправленных версий) вы можете посетить сайт www.electrotank.com/ElectroServer.

Примечание: *В папке AppendixB на приложенном CD-ROM-диске вы найдете два чата, написанных с помощью объекта ElectroServerAS. Chat_fullfeatured.fla обладает всеми особенностями хорошего чата. Chat_bare-bones.fla содержит минимальный набор особенностей, необходимых для работы чата.*

Перетаскивание действий (Click-and-Drag)

Мы упомянули, что в объекте ElectroServerAS много методов и атрибутов, и большая часть из них очень полезна для написания игр. Тем не менее вам не придется запоминать их синтаксис. Вы можете установить эти действия прямо в Actions toolbox во Flash-панели Actions (получив очень много действий!), и перетаскивать их оттуда по мере необходимости в панель сценария. Такая установка также допускает выделение цветом всех действий ElectroServerAS.

1. Открываете Flash.
2. Находите файл install.swf в папке AppendixB и открываете его в окружении Flash (как SWF, не импортируете (import)!).
3. Нажимаете кнопку Install ElectroServerAS Object Actions. После этого действия должны быть установлены. Закройте SWF и взгляните на панель Actions. Вы должны увидеть в качестве одной из опций объект ElectroServerAS. В данный момент с действиями не устанавливается никакой справочной документации; в качестве документации служит данное приложение. Однако к моменту опубликования данной книги уже наверно появится новая версия инсталлятора действий с документацией (на сайте www.electrotank.com/ElectroServer).
4. Для того чтобы выделение действий цветом вступило в силу, закройте Flash и откройте снова.

Для того чтобы все эти установленные действия работали, вам нужно также включить файл ElectroServerAS.as в соответствующий сценарий (использующий данные действия). ElectroServerAS.as - это текстовый файл с кодом ActionScript, определяющий методы и атрибуты объекта ElectroServerAS. Вам даже не нужно открывать или редактировать этот файл (если вы не собираетесь расширить возможности объекта ElectroServerAS).

Вам нужно включить файл ElectroServerAS.as в вашу программу Flash с помощью следующих шагов:

1. Скопируйте ElectroServerAS.as в директорию, в которой находится ваша программа Flash.
2. Скопируйте в эту же директорию файл WDDX_mx.as.
3. На основной монтажной линейке в каждом месте, где будет использоваться объект ElectroServerAS, добавьте.

```
#include "ElectroServerAS.as"
```

Это все. Теперь при создании файла SWF в него будет включена вся информация из файла ElectroServerAS.as. Вам не нужно самим включать файл WDDX_mx.as, так как ElectroServerAS.as сделает это автоматически.

Методы и атрибуты объекта ElectroServerAS

А теперь давайте рассмотрим по порядку все методы и атрибуты объекта ElectroServerAS.

Примечание: *Обратите внимание, что все перечисленные ниже методы рассчитаны на работу с Flash Player 6; именно Flash Player 6 является версией, поддерживающей и правильно интерпретирующей эти методы.*

Примечание: *ElectroServer работает как с Flash 5, так и с Flash MX. Однако объект ElectroServerAS работает только с Flash MX, из-за изменений в области видимости функций Flash MX.*

ElectroServerAS. addToHistory

Использование. ElectroServerAS.addToHistory(message)

Параметр. message - строка, которая будет добавлена к атрибуту history.

Возвращает. Не возвращает ничего.

Описание. Метод. Добавляет строку к атрибуту history (ElectroServerAS.history).

Пример. Добавление строки к атрибуту history:

```
ElectroServerAS.addToHistory("Anyone for a game of golf?")
```

ElectroServerAS. ban

Использование. ElectroServerAS.ban(who, why)

Параметры.

who — имя пользователя, которого вы хотите запретить.

why — причина запрета.

Возвращает. Не возвращает ничего.

Описание. Метод. Отсоединяет пользователя от сокет-сервера ElectroServer и запрещает последующее его соединение (с сервером) с данного IP-адреса. Этот метод доступен только пользователям с правами администратора. См. также ElectroServerAS.login.

Пример. Запрещение пользователя:

```
ElectroServerAS.ban("meanie", "Offensive language")
```

ElectroServerAS. cancelChallenge

Использование. ElectroServerAS.cancelChallenge().

Параметры. Нет.

Возвращает. Не возвращает ничего.

Описание. Метод; Отменяет посланный ранее вызов на игру. На стороне пользователя, которому послан вызов, создается событие ElectroServerAS.challengeCancelled.

ElectroServerAS. challengee

Использование. ElectroServerAS.challenge(who, game)

Параметры.

who - имя пользователя, которому вы хотите послать вызов.

game - имя игры,

Возвращает. Не возвращает ничего.

Описание. Метод. Посылает вызов на игру. На стороне пользователя, которому послан вызов, создается событие ElectroServer AS. challengeReceived. Атрибуту ElectroServerAS.challenging присваивается значение true.

Пример. Отправка вызова:

```
ElectroServerAS.challenge("jobem","Mini Golf")
```

ElectroServerAS. challengeAnswered

Использование. ElectroServerAS.challengeAnswered(which)

Параметр. which - строка с ответом на вызов ("accepted", "declined", или "autodeclined").

Возвращает. Не возвращает ничего.

Описание. Метод; callback функция, выполняемая объектом ElectroServerAS тогда, когда пользователь отвечает на присланный ему вызов. Если пользователь принимает вызов, функции передается "accepted". В случае отказа - "declined". Ec-

ли пользователь задал автоматическое отклонение вызова на игру, функции передается "autodeclined".

Пример. Ниже приведен пример создания функции для выполнения при ответе на присланный вызов:

```
function challengeAnswered(which) {
if (which == "accepted") {
root.gotoAndStop("game");
} else if (which == "declined") {
trace("declined");
} else if (which == "autodeclined") {
trace("auto declined");
}
}
ElectroServerAS.challengeAnswered = this.challengeAnswered;
```

ElectroServerAS. challengeCancelled

Использование. ElectroServerAS.challengeCancelled

Параметры. Нет.

Возвращает. Не возвращает ничего.

Описание. Метод. Callback-функция, выполняемая объектом ElectroServerAS в том случае, когда посланный вам вызов на игру был отменен.

ElectroServerAS. challengeReceived

Использование. ElectroServerAS.challengeReceived(who, game)

Параметры.

who - имя пользователя, пославшего вам вызов.

game - имя игры.

Возвращает. Не возвращает ничего.

Описание. Метод. Callback-функция, выполняемая объектом ElectroServerAS в случае получения вызова на игру. Вызов на игру вам могут прислать другие пользователи.

ElectroServerAS. challenging

Использование. ElectroServerAS.challenging

Описание. Атрибут. Булево значение. Если true, значит вызов был послан, а ответ еще не получен. Если в это время вам придет вызов, то он будет автоматически отклонен. В противном случае (данный атрибут равен false) вы можете получать вызовы от других пользователей. Этот атрибут используется внутри объекта ElectroServerAS.

ElectroServerAS. chatReceiver

Использование. ElectroServerAS.chatReceiver(info)

Параметры. info - объект, содержащий атрибуты from, type и body.

Возвращает. Не возвращает ничего.

Описание. Метод. Callback-функция, выполняемая объектом ElectroServerAS в случае получения чат-сообщения. При выполнении этого метода ему передается объект с атрибутами from, type, и body. Атрибут from содержит имя автора сообщения. Атрибут type указывает тип сообщения. Тип "public" означает сообщение для всей комнаты; тип "private" -личное сообщение. Атрибут body содержит само сообщение.

Пример. Ниже приведен пример создания функции для выполнения при получении сообщения.

```
function messageArrived(info) {
    var from = info.from;
    var type = info.type;
    var body = info.body;
    if (type == "public") {
        var msg = from+": "+body+newline;
    } else if (type == "private") {
        var msg = from+"(private): "+body+newline;
    }
chat.window.text = ES.addToHistory(msg);
chat.bar.setScrollPosition(chat.window.maxscroll);
}
ElectroServerAS.chatReceiver = this.messageArrived;
```

ElectroServerAS. connectToServer

Использование. ElectroServerAS.connectToServer()

Параметры. Нет.

Возвращает. Не возвращает ничего.

Описание. Метод. Использует атрибуты ElectroServerAS.port и ElectroServerAS.ip и инициализирует соединение с сокет-сервером ElectroServer. Смотри также ElectroServerAS.port и ElectroServerAS. ip.

Пример. Установление соединения с сокет-сервером ElectroServer:

```
ElectroServerAS.connectToServer()
```

ElectroServerAS. createVariable

Использование. ElectroServerAS.createVariable(name, value, deleteOnExit, lock)

Параметры.

name - имя серверной переменной, которую вы хотите создать в данной комнате.

value - значение переменной (строка).

deleteOnExit - либо true (или "True"), либо false (или "False"). Если true, переменная удаляется в момент вашего выхода из комнаты (в противном случае не удаляется)

lock - либо true (или "True"), либо false (или "False"). Если true, переменная не может быть модифицирована (в противном случае может). Переменная может быть удалена с помощью ElectroServerAS.deleteVariable() независимо от значения lock.

Возвращает. Не возвращает ничего.

Описание. Метод. Создает или модифицирует переменную в вашей текущей комнате на сокет-сервере. В случае создания, модификации или удаления переменной об этом сообщается всем в комнате с помощью события ElectroServerAS.roomVariablesChanged. Все эти переменные хранятся в объекте под названием roomVars, находящемся в объекте ElectroServerAS.

Пример. Создание переменной комнаты:

```
ElectroServerAS.createVariable("secret_door","door3",true,false)
```

ElectroServerAS. deleteVariable

Использование. ElectroServerAS.deleteVariable(name)

Параметры. name - имя переменной, которую вы хотите удалить.

Возвращает. Не возвращает ничего.

Описание. Метод. Удаляет переменную комнаты с заданным именем. Переменная удаляется даже если она заблокирована (См. ElectroServerAS.createVariable()). Об удалении посылается сообщение всем в комнате.

Пример. Удаление переменной комнаты:

```
ElectroServerAS.deleteVariable("secret_door")
```

ElectroServerAS.disconnectFromServer

Использование. ElectroServerAS.disconnectFromServer()

Параметры. Нет.

Возвращает. Не возвращает ничего.

Описание. Метод. Закрывает соединение между Flash и сокет-сервером ElectroServer.

Пример. Отсоединение Flash-клиента от сокет-сервера ElectroServer:

```
ElectroServerAS.disconnectFromServer()
```

ElectroServerAS. getHistory

Использование. ElectroServerAS.getHistory()

Параметры. Нет.

Возвращает. Строку ElectroServerAS.history.

Описание. Метод; возвращает историю чата. История чата хранится в виде строки в ElectroServerAS.history и может быть добавлена с помощью функции ElectroServerAS.addToHistory().

Пример. Получение истории чата:

```
myHistory = ElectroServerAS.getHistory()
```

ElectroServerAS.getRoomList

Использование. ElectroServerAS.getRoomList()

Параметры. Нет.

Возвращает. Массив объектов.

Описание. Метод. Возвращает массив объектов. Каждый объект в массиве описывает одну комнату и имеет два атрибута: name и total. Атрибут name содержит имя комнаты; атрибут total - общее число участников в комнате.

Пример. Приведенный ниже ActionScript-код проходит по списку комнат и в окне вывода показывает для каждой комнаты имя комнаты и количество участников:

```
var theRooms = ElectroServerAS.getRoomList();
for (i in theRooms) {
trace(theRooms[i].name);
trace(theRooms[i].total);
}
```

ElectroServerAS.getUserList

Использование. ElectroServerAS.getUserList()

Параметры. Нет.

Возвращает. Массив объектов.

Описание. Метод. Каждый объект в массиве описывает одного пользователя и имеет один атрибут: name. Атрибут name содержит имя пользователя из вашей комнаты.

Пример. Приведенный ниже ActionScript код проходит по списку пользователей вашей комнаты:

```
var theUsers = ElectroServerAS.getUserList();
for (i in theUsers) {
trace(theUsers[i] .name);
}
```

ElectroServerAS.history

Использование. ElectroServerAS.history

Описание. Атрибут; в нем хранится история чата в виде строки. Сейчас метод ElectroServerAS.getHistory() просто возвращает этот атрибут. Однако в будущих версиях объекта ElectroServerAS история чата может храниться в другом виде, так что для доступа к истории рекомендуется использовать метод ElectroServerAS.getHistoryO.

Пример. Следующая строчка демонстрирует использование данного атрибута:

```
myHistory = ElectroServerAS.history
```

ElectroServerAS.inGame

Использование. ElectroServerAS.inGame

Описание. Атрибут; это булево значение (true или false). Если true, вы в данный момент принимаете участие в игре. Если вы получаете вызов на игру и данный атрибут равен true (в игре), то вызов автоматически отклоняется. Этот атрибут используется внутри объекта ElectroServerAS.

ElectroServerAS.ip

Использование. ElectroServerAS.ip

Описание. Атрибут; в нем хранится IP-адрес сервера, с которым вы хотите соединиться. Этот атрибут (так же как и ElectroServerAS.port) должен быть задан для корректной работы метода ElectroServerAS.connectToServer(). IP-адрес может быть в численном или текстовом виде (например, "23.244.81.5" или "macromedia.com").

Пример.

```
ElectroServer = new ElectroServerAS();
ElectroServerAS.ip = "localhost";
ElectroServerAS.port = 8080;
ElectroServerAS.connectToServer();
```

ElectroServerAS. isResponding

Использование. ElectroServerAS. isResponding

Описание. Атрибут; это булево значение (true или false), используемое внутри объекта ElectroServerAS. Значение true присваивается в момент получения вызова на игру. В то время, пока этот атрибут равен true, все последующие вызовы автоматически отклоняются. Как только вы ответите на первый вызов (примете его или отклоните), данному атрибуту присваивается false.

ElectroServerAS.joinRoom

Использование. ElectroServer AS.joinRoom(name)

Параметры. name-имя комнаты, в которую вы хотите войти.

Возвращает: Не возвращает ничего.

Описание. Метод; меняет вашу комнату на комнату, указанную в параметре name. Если указанная комната еще не существует, она создается. Имя комнаты, в которую вы хотите войти, сохраняется в атрибуте ElectroServerAS.myRoom.

Пример. Вход в комнату под названием "Lobby":

```
ElectroServerAS.joinRoom("Lobby")
```

ElectroServerAS. kick

Использование. ElectroServerAS.kick(who,why)

Параметры.

who—имя пользователя, которого вы хотите удалить.

why—причина удаления.

Возвращает. Не возвращает ничего.

Описание. Метод; отсоединяет пользователя от сокет-сервера ElectroServer. Этот метод доступен только пользователям с правами администратора.

Пример. Удаление пользователя с сервера:

```
ElectroServerAS.kick("meanie")
```

ElectroServerAS.leaveAlone

Использование. ElectroServerAS.leaveAlone

Описание. Атрибут; это булево значение (true или false); по умолчанию false. Если true, то все вызовы на игру автоматически отклоняются.

ElectroServerAS.login

Использование. ElectroServer AS .challenge(name, password)

Параметры.

name—имя пользователя, под которым вы хотите войти.

password—необязательный параметр с паролем.

Возвращает. Не возвращает ничего.

Описание. Метод; позволяет пользователю войти на сервер. Если используется пароль, то имя и пароль сравниваются с именами и паролями пользователей с правами администраторов. В противном случае пользователь просто заходит на сервер. Пользователь с правами администратора создается с помощью вспомогательной административной программы сокет-сервера ElectroServer.

Пример. Вход пользователя на сервер:

```
ElectroServerAS.login("jobem")
```

Вход на сервер пользователя с правами администратора:

```
ElectroServerAS.login("important_person","his_password")
```

ElectroServerAS.loginResponse

Использование. ElectroServerAS.loginResponse(success, reason)

Параметры.

success— это булево значение (true или false). Если true, то вход на сервер произошел успешно, в противном случае войти на сервер не удалось.

reason—Строка с сообщением о причине неудачи (в случае неудачной попытки входа).

Возвращает. Не возвращает ничего.

Описание. Метод; callback-функция, выполняемая объектом ElectroServerAS после получения с сервера ответа на попытку войти. Первый параметр, success, равен true в случае успеха. В случае неудачи (success равен false) передается также второй параметр (reason) с причиной неудачи.

Пример. Ниже приведен пример создания функции для выполнения при получении ответа на попытку войти:

```
function loginResponse(success, reason) {
if (success) {
ElectroServerAS.joinRoom("Lobby");
chat.gotoAndStop("chat");
} else {
trace("reason="+reason);
}
}
ElectroServerAS.loginResponse = this.loginResponse;
```

ElectroServerAS. moveReceived

Использование. ElectroServerAS.moveReceived(object)

Параметры. object — Некоторый объект, созданный другим игроком.

Возвращает. Не возвращает ничего.

Описание. Метод; callback функция, выполняемая объектом ElectroServerAS при получении хода от противника. Передаваемый объект может содержать любой тип данных, включая массивы, XML-объекты и переменные.

Пример. Ниже приведен пример создания функции для выполнения при получении хода. Эта функция просто вызывает функцию trace для трассировки имен и значений всех атрибутов переданного объекта:

```
function moveReceived(ob) {
for (i in ob) {
trace(i+"="+ob[i]);
}
}
ElectroServerAS.moveReceived = this. moveReceived;
```

ElectroServerAS. my Room

Использование. ElectroServerAS.myRoom

Описание. Атрибут; в нем в виде строки хранится имя комнаты, в которой вы находитесь в данный момент.

ElectroServerAS. onConnection

Использование. ElectroServerAS.onConnection(success)

Параметры.

success — это булево значение (true или false). Если true, то соединение с сервером призошло успешно, в противном случае соединиться с сервером не удалось.

Возвращает. Не возвращает ничего.

Описание. Метод; callback-функция, выполняемая объектом ElectroServerAS после установления и проверки соединения с сокет-сервером ElectroServer. В случае успеха передаваемый параметр success равен true (в противном случае false). Перед выполнением этой функции сервер должен также отправить тестовое сообщение, проверяющее работоспособность соединения.

Пример. Ниже приведен пример создания функции для выполнения после установления и проверки соединения с сокет-сервером ElectroServer:

```
function connectionResponse(success) {
if (success) {
chat.gotoAndStop("login");
} else {
trace("connection failed");
}
}
ElectroServerAS.onConnection = this.connectionResponse;
```

ElectroServerAS.onPlayersInRoomChange

Использование. ElectroServerAS.onPlayersInRoomChange(num)

Параметры. num — число участников в вашей комнате.

Возвращает. Не возвращает ничего.

Описание. Метод; callback-функция, выполняемая объектом ElectroServerAS при изменении числа участников в вашей комнате (в момент прихода или ухода кого-либо).

Пример. Ниже приведен пример создания функции для выполнения при изменении числа участников в вашей комнате. Если в комнате два человека, то функция начинает игру (в предположении игры на двух человек):

```
function numPlayers(num) {
if (num == 2) {
startGame();
}
}
ElectroServerAS.onPlayersInRoomChange = this.numPlayers;
```

ElectroServerAS.onRoomVarChange

Использование. ElectroServerAS.onRoomVarChange(roomVars, type, name)

Параметры.

roomVars — Объект с переменными комнаты.

type — строка с типом изменения ("list", "update",или "delete").

name — имя измененной переменной.

Возвращает. Не возвращает ничего.

Описание. Метод; callback-функция, выполняемая объектом ElectroServerAS либо при изменении любой переменной из списка переменных комнаты (переменные, ассоциированные с комнатой и хранящиеся на сервере), либо при вашем первом входе в комнату. При вашем первом входе в комнату вам посылается список всех переменных данной комнаты (в этом случае параметр type равен "list"). При создании или модификации переменной параметр type равен "update", а параметр name содержит имя соответствующей переменной. При удалении переменной параметр type равен "delete", а параметр name содержит имя удаленной переменной.

ElectroServerAS. opponent

Использование. ElectroServerAS.opponent

Описание. Атрибут; в нем хранится имя вашего противника. Этот атрибут создается в тот момент, когда вы приняли вызов на игру.

ElectroServerAS.player

Использование. ElectroServerAS.player

Описание. Атрибут; в нем хранится ваш "номер игрока" в игре. Если вы участвуете в игре на двух игроков, то значение этого атрибута может быть 1 или 2.

Пример. Ниже приведен пример возможного кода в начале шахматной игры:

```
if (ElectroServerAS.player == 1) {
myChessPieceColor= "white";
} else if (ElectroServerAS.player == 2) {
myChessPieceColor = "black";
}
```

ElectroServerAS. port

Использование. ElectroServerAS.port

Описание. Атрибут; в нем хранится порт сервера, с которым вы хотите соединиться. Этот атрибут (так же, как и ElectroServerAS.ip) должен быть задан для корректной работы метода ElectroServerAS.connectToServer().

Пример.

```
ElectroServer = new ElectroServerAS();
ElectroServerAS.ip="localhost";
ElectroServerAS.port=8080;
ElectroServerAS.connect();
```

ElectroServerAS. roomListChanged

Использование. ElectroServerAS.roomListChanged(rooms)

Параметры.

rooms — массив объектов.

Возвращает. Не возвращает ничего.

Описание. Метод; callback-функция, выполняемая объектом ElectroServerAS при изменении списка видимых комнат. При выполнении этой функции передается массив объектов. Каждый объект в массиве описывает одну комнату и имеет два атрибута: name и total. Атрибут name содержит имя комнаты; атрибут total - общее число участников в комнате.

Пример. Ниже приведен пример создания функции для выполнения при изменении списка видимых комнат. Данная функция выводит полученную информацию в текстовое поле, в формате типа Lobby(32):

```
function roomListChanged(roomList) {
roomList.text = "";
for (var i = 0; i<roomList.length; ++i) {
chat.roomList.text+=
roomList[i].name+"("+roomList[i].total+")"+newline;
}
}
ElectroServerAS.roomListChanged = this.roomListChanged;
```

ElectroServerAS. rooms

Использование. ElectroServerAS.rooms

Описание. Атрибут; массив, хранящий информацию о каждой комнате. Каждый объект в массиве описывает одну комнату и имеет два атрибута: name и total. Атрибут name содержит имя комнаты; атрибут total - общее число участников в комнате. Для получения этого массива рекомендуется использовать метод ElectroServerAS.getRoomList(). В данный момент этот метод просто возвращает данный атрибут, но в следующих версиях объекта ElectroServerAS информация о комнатах может храниться в другом виде.

ElectroServerAS. roomVars

Использование. ElectroServerAS.roomVars

Описание. Атрибут; объект, хранящий переменные, ассоциированные с комнатой. Каждый раз при создании, модификации или удалении переменной происходит также обновление данного объекта. Для получения информации об этом вам нужно создать обработчик события ElectroServerAS.onRoomVarChange.

Пример. Приведенный ниже пример показывает, как вывести имена и значения всех переменных комнаты:

```
ob = ElectroServerAS.roomVars;
for (i in ob) {
trace(i+"="+ob[i]);
}
```

ElectroServerAS.sendData

Использование. ElectroServerAS.sendData(msg)

Параметры. msg — данные, которые вы хотите отправить на сервер.

Возвращает. Не возвращает ничего.

Описание. Метод; посылает на сервер содержание параметра msg. Этот метод используется внутри объекта ElectroServerAS и вряд ли может понадобиться для чего-либо, кроме расширения объекта ElectroServerAS.

ElectroServerAS.sendMessage

Использование. ElectroServerAS.sendMessage(msg, who)

Параметры.

msg — чат-сообщение, которое вы хотите послать.

who — строка, указывающая, кому вы хотите послать сообщение. Если данный параметр равен "АИ", то сообщение посылается всем в вашей текущей комнате. Если указано имя пользователя (в любой комнате), то сообщение посылается лично.

Возвращает. Не возвращает ничего.

Описание. Метод; посылает чат-сообщение пользователю или всей комнате. Именно этот метод используется для нормального чата и личных сообщений.

Пример. Отправка сообщения всей комнате:

```
ElectroServerAS.sendMessage("Good morning Raleigh!","All")
```

Отправка личного сообщения пользователю "jobem":

```
ElectroServerAS.sendMessage("Hey man, where've you been?","jobem")
```

ElectroServerAS.sendMove

Использование. ElectroServerAS. sendMove(who, what)

Параметры.

who — имя пользователя, которому вы хотите послать ход.

what — посылаемый объект.

Возвращает. Не возвращает ничего.

Описание. Метод; посылает объект указанному пользователю. Объект имеет тип object и может содержать любые другие объекты данных, такие как массивы, XML-объекты и.т. д. С помощью этого метода осуществляется ход в игре.

Пример. Отправка хода игроку "jobem":

```
myObject=new Object();
myObject.ball_x=32;
myObject.ball_y=413;
ElectroServerAS.sendMove("jobem",myObject);
```

ElectroServerAS.sendSystemMessage

Использование. ElectroServerAS.sendSystemMessage(msg)

Параметры. who — сообщение (строка), которое вы хотите послать.

Возвращает. Не возвращает ничего.

Описание. Метод; посылает сообщение всем пользователям на сервере (в каждой комнате). Этот метод доступен только для пользователей с правами администратора.

Пример. Следующая строчка ActionScript посылает всем сообщение:

```
ElectroServerAS.sendSystemMessage("The server is about to be rebooted. Please refresh in 1 minute.")
```

ElectroServerAS.userListChanged

Использование. ElectroServerAS.userListChanged(users)

Параметры. users — массив объектов.

Возвращает. Не возвращает ничего.

Описание. Метод; callback-функция, выполняемая объектом ElectroServerAS при изменении списка пользователей в вашей комнате. Это событие происходит также при вашем первом входе в комнату - вам посылается список всех пользователей в комнате. Параметр users содержит список объектов. Каждый объект представляет одного пользователя и содержит один атрибут (name) с именем пользователя.

Пример. Ниже приведен пример создания функции для выполнения при изменении списка пользователей в комнате. Для отображения списка пользователей используется Flash компонент ListBox:

```
function userListChanged(userList) {
var path = chat.userList;
path.removeAll();
for (var i = 0; i<userList.length; ++i) {
path.addItem(userList[i].name);
}
}
ElectroServerAS.userListChanged = this.userListChanged;
```

ElectroServerAS.username

Использование. ElectroServerAS.username

Описание. Атрибут; имя, под которым вы вошли на сервер.

ElectroServerAS.users

Использование. ElectroServerAS.users

Описание. Атрибут; массив объектов. Каждый объект представляет одного пользователя и содержит один атрибут (name) с именем пользователя. Для получения списка пользователей рекомендуется использовать функцию ElectroServerAS.ge-

tUserList(). В будущих версиях объекта ElectroServerAS список пользователи может храниться в другом виде.

ElectroServerAS. usersInMyRoom

Использование. ElectroServerAS .usersInMyRoom

Описание. Атрибут; целое число - общее число пользователей в вашей комнате.

new ElectroServerAS()

Использование. new ElectroServerAS()

Параметры. Нет.

Возвращает. Не возвращает ничего.

Описание. Конструктор; создает новый объект ElectroServerAS. Перед вызовом какого-либо из методов объекта ElectroServerAS вам нужно использовать данный конструктор для создания экземпляра объекта ElectroServerAS.

Пример. Создание экземпляра объекта ElectroServerAS под именем ES:

```
ES = new ElectroServerAS();
```

Предметный указатель

D

E

F

G

H

I

J

K

L

M

N

O

P

В

Г

Д

З

И

К

Л

М

Н

О

П

Содержание

ИЗДАТЕЛЬСТВО

«КУДИЦ-ОБРАЗ»

Тел.: (095) 333-82-11; ok@kudits.ru, http://books.kudits.ru

ПРИОБРЕТАЙТЕ КНИГИ У НАШИХ ПАРТНЕРОВ

Барнаул
Социалистический пр-т, 117а,
(3852) 38-18-72

Великий Новгород
ул. Б. Санкт-Петербургская, 44
тел. (81622) 73-188 доб. 34

Екатеринбург
"Книжный мир",
ул. 8 Марта, 8г, (3432) 71-18-87
ООО "КДК Дом книги"
ул. Блюхера, 51

Краснодар
"Мир книги", ул. Буденного, 147
"Колос", ул. Красная, 100,
тел. (8612) 59-41-32

Москва
"ОПТИМА+"
(095) 333-65-67,
ok@kudits.ru
http://books.kudits.ru

Новосибирск
"Книжный пассаж",
ул. Ленина, 10а, (3832) 29-50-30
"Сибирский Дом Книги",
Красный пр-т, 153, (3832) 26-62-39
"Книжный мир", пр-т К.Маркса, 51

Омск
"Книжный Мир",
ул. Ленина, 17/19, (3812) 24-32-54

Ростов-на-Дону
"Мир книги", Ворошиловский пр-т, 33,
(8632) 62-54-61

Санкт-Петербург
"Санкт-Петербургский Дом книги"
Невский проспект, 28, тел. (812) 312-01-84
издательство "Наука и Техника"
пр. Обуховской Обороны, 107,
тел. (812) 567-70-25, 567-70-26

Саратов
"Книжный Мир",
пр-т Кирова, 32, (8452) 32-98-14

Ставрополь
"Книжный Мир", ул. Мира, 337, (8652) 35-47-90

Таганрог
"Компьютерная книга", ул. Чехова, 31,
тел. (8634) 37-13-12

Томск
"Книжный Мир", ул. Ленина, 141, (3822) 51-07-16

Уфа
000 ПКП "Азия", тел./факс: (3472) 50-39-00
Оптовая торговля
Ул. Зенцова, 70
Розничная торговля
Магазин "Оазис", ул. Чернышевского, 88
Магазин "Книжник", пр. Октября, 106

Ханты-Мансийск
Магазин "Книги", ул. Ленина, 39

Челябинск
"Книжный Мир", ул. Кирова, 90, (3512) 33-19-58

Ярославль
Магазин "Наука",
ул. Володарского, 63, (0852) 25-95-04

ЗАКАЗ КНИГ НАЛОЖЕННЫМ ПЛАТЕЖОМ

Издательство «ОЦ КУДИЦ-ОБРАЗ» осуществляет рассылку книг по почте.

Заказы принимаются по адресу: 121354, Москва, а/я 18; или по e-mail: ok@kudits.ru
Условия рассылки: Сумма наложенного платежа складывается из оптовой цены книг, накладных расходов «ОЦ КУДИЦ-ОБРАЗ» на пересылку (30% от стоимости книг) и почтовых расходов (по тарифам почты РФ).

Заказы из регионов России с авиадоставкой, а также заказы из стран ближнего и дальнего зарубежья обслуживаются только по предварительной оплате.

ПОПУЛЯРНЫЕ WEB-ПРИЛОЖЕНИЯ на FLASH MX

Книга Популярные Web-приложения на FLASH MX написана для разработчика приложений в Macromedia FLASH, желающего создавать полномасштабные программы, решающие реальные проблемы. Она написана ведущими представителями сообщества программистов Flash, и каждая глава в первую очередь сосредоточена на приемах программирования. Хотя любую главу можно читать независимо от других, вместе они образуют наиболее полный справочник приёмов и методов программирования в Macromedia Flash.

Сосредоточившись на реальных проектах и приложениях, авторы показывают вам, как написать на основе интернет-технологий систему опросов, почтового клиента и программу мгновенного обмена сообщениями, дискуссионный форум, аватар чат, видео чат, многопользовательскую игру и многое другое!

Так как разработчики пользуются многими языками, проекты в данной книге написаны с помощью Java, ColdFusion и ASP.NET; используются также C# и немного PHP. Клиентская часть (Macromedia Flash) приложений написана таким образом, что вы можете всегда использовать тот же самый клиент независимо от выбранного языка на стороне сервера.

Является наиболее полным справочником приемов и методов программирования

macromedia PRESS macromedia MX

Книги Macromedia Press публикуются в сотрудничестве с Peachpit Press

Peachpit Press
1249 Eighth Street, Berkeley, CA 94710
510-524-2178 tel
510-524-2221 fax

www.peachpit.com
www.macromedia.com

КУДИЦ-ОБРАЗ
тел./факс: (095) 333-82-11,333-65-67
E-mail: ok@kudits.ru
http://books.kudits.ru
121354; Москва, а/я 18, "КУДИЦ-ОБРАЗ"

ISBN 5-93378-079-0