RESTGroups – User Guide

Release 1.0

Tadeusz Kobus, Paweł T. Wojciechowski
www.it-soa.pl/restgroups

January 2011

Contents

1 Overview 3

2 Requirements 5

3 Compilation and configuration 7

4 Example application: Service replication 9

5 Programming Interface 11

6 Running a demo program 17

7 Bibliography 19

Bibliography 21

Contents:

CHAPTER 1

Overview

RESTGroups is a group communication API for developing resilient RESTful Web Services. Our current implementation is based on Spread Toolkit [SC] [AS98] and provides the programming primitives for:

- detection of malfunctioning/crashed processes,
- reliable point-to-point transfer of messages,
- formation of processes into groups, the structure of which can change at runtime,
- reliable message multicasting with a wide range of guarantees concerning delivery of messages to group members (e.g. causally-, fifo- and totally- ordered message delivery).

Contrary to Spread and other group communication systems, RESTGroups represents group communication abstractions as resources on the Web, addressed by URIs [BFLM98], and has a small API that consists of just four methods of the HTTP protocol:

- GET is used to retrieve data (for example, a message) or perform a query on a resource; the data returned from the RESTGroups service is a representation of the requested resource;
- POST is used to create a new resource (for example, to extend a process group with a new process or to send/broadcast a new message); the RESTGroups service may respond with data or a status indicating success or failure;
- PUT is used to update existing resources or data;
- DELETE is used to remove a resource or data (for example, to remove a process from a process group). (In some cases, the update and delete actions may be performed with POST operations as well.)

This provides Web application developers with a uniform way of using group communication, aiding software integration and reuse. The applications can be implemented with the use of a variety of programming languages and can run on different platforms. They can also conform to the REST architectural style [F00] [FT02].

CHAPTER 2

Requirements

RESTGroups requires the following components to be installed on the target system:

- execution platform: Java Runtime Environment 1.6
- compilation platform: Java Software Development Kit 1.6
- additional libraries:
  - JDom (Apache-style open source license with the acknowledgment clause removed)
  - RESTlet (LGPL v3, LGPL v2.1, CDDL v1.0, EPL v1.0)
  - Spread Toolkit (Custom two license model)
  - Xerces (Apache Licence v1.1)

CHAPTER 3

Compilation and configuration

The compilation of RESTGroups requires all of the listed in Requirements libraries. Appropriate jar archives are placed in the lib directory.

The project can be build using Apache Ant. The follwing ant targets are available:

- [default] - compiles sources, generates documentation (see below) and creates jar archive,
- `clean` - removes `bin`, `dist`, `javadoc` directories,
- `compile_user` - compiles demo application's source files to the bin directory,
- `compile_server` - compiles server's source files to the `bin` directory,
- `docs` - generates the documentation (sphinx in html format, javadocs); documenation to the project is placed in the `docs` directory whereas the *javadoc* documentation is located in the `javadoc` directory,
- `jar` - creates jar archive and places it in the `dist` directory,
- `zip` - creates the zip archive of the project and places it in the project's main directory,
- `zipnosource` - creates the zip archive of the project with jar archive but no sources and places it in the project's main directory,

In order to run a target execute the `ant <target>` command in the project's base directory. Calling `ant` without any arguments invokes the default target.

Before the RESTGroups servers can start the Spread Toolkit has to be setup on all the machines where the Spread daemon is supposed to be running. In order to start the RESTGroups server a proper XML document containing the RESTGroups settings have to be created. A sample `settings.xml` configuration file can be found in the `utils` directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<restgroups>
  <restgroupsIP>127.0.0.1</restgroupsIP>
  <restgroupsPort>8182</restgroupsPort>
  <schemas>
    <clientMessage>../schemas/clientMessage.xsd</clientMessage>
    <clientMessageSingleGroup>../schemas/clientMessageSingleGroup.xsd
    </clientMessageSingleGroup>
    <profileMessage>../schemas/profileMessage.xsd</profileMessage>
    <connectMessage>../schemas/profilesPilotMessage.xsd</connectMessage>
```

```
  </schemas>
  <engine>
    <spread factoryClassName="gcsImplementations.spreadGcs.SpreadGcsFactory">
      <daemonIP>127.0.0.1</daemonIP>
      <daemonPort>4803</daemonPort>
    </spread>
  </engine>
</restgroups>
```

As one can see the configuration file contains such information as:

- the IP address and port the RESTGroups server will be running on,
- location of the appropriate XML schema files used to validate the incoming messages,
- the IP address and port of the Spread daemon the RESTGroups server will be using.

Once the configuration file is complete the RESTGroups server can be started by executing the following command:

```
java -cp "RestGroups.jar:jdom.jar:org.restlet.ext.xml.jar:
  org.restlet.ext.servlet.jar:org.restlet.jar:spread-4.1.0.jar"
    restgroupsServer.ServerMain settings.xml
```

In the `utils` directory there is also a script `runServer.sh` which simplifies the process of starting a RESTGroups server:

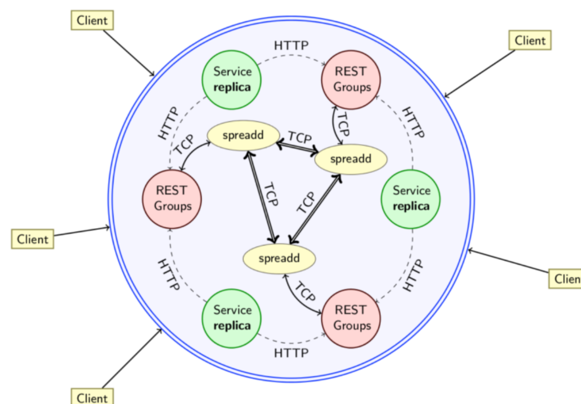
```
./runServer.sh settings.xml
```

CHAPTER 4

Example application: Service replication

An example application of RESTGroups is development of resilient Web Services. Service resilience is the continued availability of a service despite failures and other negative changes in its environment. A typical way of increasing service resilience is to replicate it [S90] [S93]. Service replication means deployment of a service on several server machines, each of which may fail independently and coordinating them using group communication. The process of replication the service should be transparent to the clients.

The Figure below shows an architecture of a replicated service using RESTGroups:



The communication infrastructure is provided by a backbone network of Spread daemons (*spreadd*) to which Spread clients connect.

RESTGroups servers act as clients for the Spread daemons. However they are stateless, i.e, they do not store any information about its clients (which are service replicas), except for the necessary GCS sessions and so called permanent connections, required for detection of client crashes (for details see Programming Interface). Importantly, the RESTGroups server does not have any representation in the group communication system, which is the back-end of RESTGroups. RESTGroups server act as a proxy between a service that want to use group communication system and the RESTGroups back-end.

A crash of a given RESTGroups server results in the disconnection of all clients of this server, which can then establish connection with another RESTGroups server. In order to tolerate failures, the REST-

Groups clients can establish connections with separate RESTGroups servers, that are available within the same group.

Replicating a service requires the following steps (given *n* machines):

- setting up from *1* up to *n* Spread daemons on different machines (yellow ellipses with the *spreadd* label),
- setting up from *1* up to *n* RESTGroups servers (red bubbles with the *RESTGroups* label); each RESTGroups server communicates only with one Spread daemon; usually RESTGroups server is deployed on the same machine where runs the Spread daemon with which the RESTGroups server communicates.
- running from 2 up to *n* instances of the replicated service (green bubbles with the *Service replica* label) on different machines. Each replica can communicate with one or many RESTGroups servers depending on the level of fault tolerance to be provided by the setup.

Each replica joins the same group so messages broadcast to the group can be delivered to all the replicas in the same order.

The client (yellow rectangle with the *Client* label) of the replicated Application Service (blue bubble) can communicate with any service replica. However, all the requests sent to any service replica are distributed among all the replicas by broadcasting a totally ordered message to the group the service replicas have joined. Thus, all the replicas receive requests in the same order, and so also process them in the same order. Given that the processing is deterministic, all the replicas change their states in the same way. In the case of crash of one/some of the replicas the client may have to issue the same request to other (operating) service replica.

CHAPTER 5

Programming Interface

In RESTGroups various primitives provided by group communication systems are represented as resources. The client can access and manipulate them by performing HTTP requests to these resources. Some requests are required to carry XML documents. XML schemas for these documents are located in the `schemas` directory.

Let us assume that the RESTGroups server is available at the address: `http://mydomain.com:8182`. The client in examples below has the `userA` identifier.

What the client should be able to do is the following:

connect to the RESTGroups server the connection with the RESTGroups server is accomplished using two re- quests to the server. The first one, called the temporary (or pilot) request, is used to ask the server to set up a resource which is representing a new communication session. The session is created using the second request, called the permanent request. The server does not respond to this request and so the connection opened to process it remains open. Breaking of the latter connection is interpreted by the server as crash of the client. Both requests should be separated in time by no more than 5 seconds; the order of the requests is irrelevant.

Connection with the RESTGroups server is identified by a unique identifier pilotConnectionToken, created with the use of UUID numbers [UUID]. To create a random UUID value in Java, it is necessary to import the `java.util.UUID` library and call a static method `randomUUID()`.

```
import java.util.UUID;
UUID value = UUID.randomUUID();
```

The UUID number created by a client is sent in XML format in the bodies of both the pilot (temporary) and the permanent requests. A pilot request may look as follows:

```
POST http://mydomain.com:8182/groups/userA/pilotConnection

<?xml version="1.0" encoding="UTF-8"?>
<restgroups>
  <pilotConnectionToken>dec7b89c-1f08-447e-952f-9c441ec92e5c
  </pilotConnectionToken>
</restgroups>
```

Processing of this request is suspended until a corresponding permanent request is received or the timeout occurs. The `schemes/profilesPilotMessage.xsd` file is used for validation of the temporary request's body.

A permanent request may contain information about client preferences, e.g. a request of discarding the group membership messages, as below.

```
POST http://mydomain.com:8182/groups/userA

<?xml version="1.0" encoding="UTF-8"?>

<restgroups>
  <pilotConnectionToken>dec7b89c-1f08-447e-952f-9c441ec92e5c
  </pilotConnectionToken>
  <groupMembership>false</groupMembership>
</restgroups>
```

The `schema/profileMessage.xsd` file is used for validation of the permanent request's body

If a new session has been created successfully, the response message to the temporary request is returned with the `204 'Success No Content'` status. The response contains:

- `sessionID` a session identification number, stored in the response 'cookie'; from now on, all requests to the RESTGroups server must include sessionID, which will allow the server to identify the clients,
- `identifier` - URI of the client's private group, stored in the response field that is used for identification; since the names of private groups must be unique across the whole group communication system, the identifier value can be different from the name of the client, which is specified in the pilot and permanent requests.

For example, the following values could be received:

- `sessionID: d10b88e7-74f3-424a-b306-c47440a818d9`
- `identifier: http://mydomain.com:8182/groups/@userA@mydomain`

If connection with the RESTGroups server failed, the following error messages can be received:

- in response to the pilot request:
  - `408 'Request Timeout'` - if one of the two requests has not been received in a predefined period of time,
  - `500 'Server Internal Error'` - if an error occurs within the RESTGroups server
- in response to the profile request:
  - `408 'Request Timeout'` - if one of the two requests has not been received in a predefined period of time,
  - `500 'Server Internal Error'` - if an error occurs within the RESTGroups server,
  - `503 'Service unavailable'` - if an error occurs while connecting to the group communication system.

disconnect from the RESTGroups server in order to disconnect from the RESTGroups server, a client sends the following request message:

```
DELETE http://myDomain.com:8182/groups/@userA@mydomain
```

and waits for the response message with the `204 'Success No Content'` status code. Subsequently, processing of the permanent request must be finalized and the response message with the `200 'OK'` status code is returned.

Processing of the DELETE requests may occasionally fail. In such cases, the server returns the following error messages:

- `400 'Client Bad Request'` - if the client identified by sessionID (which is sent in the request's 'cookie') does not have an active RESTGroups session,
- `403 'Client Forbidden'` - if the client does not have sufficient privileges, defined by the URI profile,
- `503 'Service Unavailable'` - if the error occurred during disconnection from the group communication system.

join a group to join a group named `customGroup`, a client `userA` sends the PUT request:

```
PUT http://mydomain.com:8182/groups/customGroup/members/@userA@mydomain
```

Upon successful request, the response message with the `204 'Success No Content'` status code is returned to the client. Otherwise, the server returns one of the following error messages:

- `400 'Client Bad Request'` - if the client identified by `sessionID` does not have an active RESTGroups session,
- `503 'Service Unavailable'` - if the error occurred during disconnection from the group communication system.

Detailed information about a group to which the client has joined, such as the current group view and the membershipID identifier, will be sent inside a suitable membership message.

leave a group In order to leave a group, say `customGroup`, a user named userA sends the following DELETE request:

```
DELETE http://mydomain.com:8182/groups/customGroup/members/@userA@localhost
```

Responses to this request are the same as for joining a group (see above).

send messages There are two possible ways of sending messages to a group of users or to a single user, identified by URI of the private group to which it belongs (only one user can belong to a given private group). The first way can be applied in every case; it uses a predefined resource `/multicast` and requires to specify - in the body of a message - the name of the message recipient, i.e. an identifier of a group or a user to whom the message will be sent. The second way of sending messages is convenient if a message (or messages) are addressed for a single user only - there is no need to specify the message recipient in the body of the message. However, each potential recipient of the message, i.e. a group or an individual user, must be represented by a resource, identified by URI.

Sending a message to the customGroup by referring to the `/multicast` resource, requires an XML document. The structure of this document is verified based on the `schemes/clientMessage.xsd` file which defines a proper XML schema. The following sections (or tags) of the structure must be defined:

- `guarantee` - reliability and ordering guarantees of message delivery,
- `type` - a message type,
- `groups` - a list of addresses,
- `data` - the message payload.

There are the following guarantees of message delivery:

- `unreliable` - no guarantee of message delivery,

- `reliable` - reliable broadcast,
- `fifo` - fifo broadcast (first-in-first-out),
- `causal` - causal broadcast, consistent with Lamport's definition of causality,
- `safe` - total order broadcast,
- `agreed`- total order broadcast that is consistent with causal broadcast, i.e. messages are delivered to all recipients in the same order, and the order agrees with the causal relation between messages.

```
POST http://mydomain.com:8182/multicast

<?xml version="1.0" encoding="UTF-8"?>
<restgroups>
  <messages>
    <message type="regular">
      <guarantee>safe</guarantee>
      <type>0</type>
      <groups>
        <group>customGroup</group>
      </groups>
      <data>Sample message</data>
    </message>
  </messages>
</restgroups>
```

Using the second approach for sending a message to the customGroup, requires to specify an XML document. The structure of this document is verified using the `schemes/clientMessageSingleGroup.xsd` schema file. The request should look like below:

```
POST http://mydomain.com:8182/groups/customGroup/mailbox/safe

<?xml version="1.0" encoding="UTF-8"?>
<restgroups>
  <messages>
    <message type="regular">
      <type>0</type>
      <data>Sample message</data>
    </message>
  </messages>
</restgroups>
```

Note that the request's URI refers to a private mailbox located at the specified address. The last part of the URI defines the chosen guarantee of message delivery; this guarantee can take any of the six values described above.

Upon successful message sending, the RESTGroups server returns a response message with the `204 'Success No Content'` status code. In the case of an error, the server returns:

- `400 'Client Bad Request'` - if the client with the sessionID identifier in the request's 'cookie' does not have an active RESTGroups session,
- `503 'Service Unavailable'` - if an error occurs during the disconnection from the group communication system.

check for messages In order to check if there are any unread messages waiting on the RESTGroups server, the `userA` client can send the following request:

```
GET http://mydomain.com:8182/groups/@userA@mydomain/mailbox/avaliableMessages
```

In reply, the server returns an XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<restgroups>
  <messages avaliable="true"/>
</restgroups>
```

if there is at least one message waiting to be fetched, or:

```
<?xml version="1.0" encoding="UTF-8"?>
<restgroups>
  <messages available="false"/>
</restgroups>
```

otherwise.

If the operation of checking for messages has been successful, it should be returned a response status of `200 'OK'`. Otherwise, the following error codes can be returned:

- `400 'Client Bad Request'` - if the client identified in the response's 'cookie' by sessionID, does not have an active RESTGroups session,
- `403 'Client Forbidden'` - if the client does not have permission to check the mailbox,
- `503 'Service Unavailable'` - if an error occurs during the disconnection from the group communication system.

receive messages in non-blocking way in order to download a new message, the client can send the GET request:

```
GET http://127.0.0.1:8182/groups/@userA@mydomain/mailbox/nonblocking
```

or simply:

```
GET http://127.0.0.1:8182/groups/@userA@mydomain/mailbox
```

If there are no new messages to fetch, the following message will be returned:

```
<?xml version="1.0" encoding="UTF-8"?>
<restgroups>
  <messages available="false"/>
</restgroups>
```

Otherwise, an XML document will be returned, which contains aggregated application messages that have been sent (broadcast) by the sender. Below is an example XML document of this type:

```
<?xml version="1.0" encoding="UTF-8"?>
<restgroups>
  <messages available="true">
    <message type="membership">
      <membershipInfo membershipType="regular">
        <members>
          <group>@userA@mydomain</group>
          <group>@userB@mydomain</group>
        </members>
        <group>customGroup</group>
        <groupID>2130706433 1258230577 1</groupID>
        <cause type="join">@userB@mydomain</cause>
```

```
        </membershipInfo>
      </message>
      <message type="regular">
        <guarantee>safe</guarantee>
        <sender>@userB@mydomain</sender>
        <type>0</type>
        <endianMismatch>false</endianMismatch>
        <groups>
          <group>customGroup</group>
        </groups>
        <data>Hello customGroup</data>
      </message>
    </messages>
  </restgroups>
```

In the above example, two messages are returned in one XML document. The first one is a membership message of the RESTGroups system, informing about a new member of the customGroup group, identified with `@userB@localhost`. The second one is a regular message which has been sent to the customGroup group by the `userB` client.

The server's responses to this request are the same as those previously defined for checking the availability of new messages.

More information about the structure of the returned XML document can be found in the *javadoc* documentation of RESTGroups and in the source code of RESTGroups-Client in the `restgroupsClient` package.

receive messages in blocking way the RESTGroups system offers a mechanism for blocking reception of messages. If used, performing a GET request by a client is suspended until a new message (or messages) will be received by the client. Messages are sent to a client as soon as they arrive to the RESTGroups server. In order to initiate blocking reception of messages, the client sends the following GET request:

```
GET http://mydomain.com:8182/groups/@userA@mydomain/mailbox/blocking
```

In order to stop using this feature, the client should issue the DELETE request:

```
DELETE http://mydomain.com:8182/groups/@userA@mydomain/mailbox/blocking
```

The structure of responses to the above requests is the same as in the case of non-blocking messages.

CHAPTER 6

Running a demo program

Interacting with the RESTGroups server requires only performing described above HTTP requests. Therefore the programmer is not bound to any platform or programming language. The only requirement is the availability of a libraries allowing performing HTTP requests and creating/parsing XML documents.

The project is supplied with a small demo application also written in Java language. It implements a text console, analogous to the *User* application included in Spread release. The console executed on a given machine allows the client to:

- join/leave custom groups,
- sent/broadcast messages with specific guarantees to the groups or other clients,
- receive message by using both the blocking and non-blocking modes.

In order to run the demo application the user have to execute the following command:

```
java -cp "RestGroups.jar:jdom.jar:org.restlet.ext.xml.jar:
  org.restlet.ext.servlet.jar:org.restlet.jar"
    restgroupsClient.User -u joe -d "http://127.0.0.1:8182"
```

where `joe` is the name the user will be identified in the group communication system and `http://127.0.0.1:8182` is the address on which the RESTGroups server the user wants to connect to is running.

In the `utils` directory there is also a script `runUser.sh` which simplifies the process of starting the demo application:

```
./runUser.sh -u joe -d "http://127.0.0.1:8182"
```

After starting the the demo application the brief usage description is presented.

CHAPTER 7

Bibliography

Bibliography

[S90] Fred B. Schneider (1990): Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) 22(4), pp. 299-319.

[S93] Fred B. Schneider (1993): Replication management using the state-machine approach. In: Sape Mullender, editor: Distributed Systems (2nd Ed.), ACM Press/Addison-Wesley Publishing Co., pp. 169-197.

[F00] Roy T. Fielding (2000): Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine.

[FT02] Roy T. Fielding & Richard N. Taylor (2002): Principled design of the modern Web architecture. ACM Transactions on Internet Technology (TOIT) 2(2), pp. 115-150.

[SC] Spread Concepts LLC (2006). The Spread Toolkit. http://www.spread.org/.

[AS98] Yair Amir & Jonathan Stanton (1998): The Spread wide area group communication system. Technical Report CNDS-98-4, Dep. of CS, Johns Hopkins University

[BFLM98] T. Berners-Lee, R. Fielding & L. Masinter (1998): Uniform Resource Identifiers (URI): Generic Syntax. Internet Engineering Task Force. RFC 2396.

[KW09] Tadeusz Kobus & Pawel T. Wojciechowski (2009): RESTGroups: Design and implementation of the RESTful group communication service. Technical Report TR-ITSOA-OB2-1-PR-09-6, Instytut Informatyki, Politechnika Poznanska.

[UUID] The Internet Society (2005). A Universally Unique IDentifier (UUID) URN Namespace. http://www.ietf.org/rfc/rfc4122.txt.