A 90% RESTful Group Communication Service

Tadeusz Kobus Tadeusz.Kobus@cs.put.poznan.pl
Paweł T. Wojciechowski Pawel.T.Wojciechowski@cs.put.poznan.pl

Poznań University of Technology
Institute of Computer Science
Piotrowo 2, 60-965 Poznań, Poland

Technical Report RA-2/10

Abstract

In this paper we describe a 90% RESTful group communication service that we have developed for Web applications. Our system is based on Spread—a popular group communication toolkit which delivers many useful programming abstractions, such as various reliable ordered broadcasts; they can be used, e.g. for implementing resilient servers by replication. Contrary to Spread and many other such systems available as libraries of programming languages, we represent group communication abstractions as resources on the Web, addressed by URIs. To our best knowledge, this is the first approach to engineering group communication systems in this way.

1 Introduction

The *Web* can provide a common, language-independent platform for interoperable resilient services that work together to create seamless and robust systems. *Service resilience*, defined as the continued availability of a service despite failures and other negative changes in its environment, is vital in the *Service-Oriented Architecture (SOA)*. We must ensure that each service is highly available regardless of unpredictable conditions, such as sudden and significant degradation of network latency or failure of dependant services. In this paper, we describe our work on *group communication* service which can be used for implementing resilient Web services, based on REST [9, 8].

A typical way of increasing service resilience is to replicate it. *Service replication* means deployment of a service on several server machines, each of which may fail independently, and coordination of client interactions with service replicas. Each service replica, called a *process*, starts in the same initial state and executes the same requests in the same order. Thus, the replicated service executes simultaneously on all machines. A client can use any single response to its request to the service. A replicated service is available continuously, tolerating crashes of individual server machines. If required, these machines can be located in geographically distant places, connected via a wide-area network.

A general model of such replication is called *replicated state machine* [18, 17]. Two properties are guaranteed by the replicated state machine: (1) each non-fault replica receives every request (the *agreement property*), and (2) each non-fault replica processes the requests in the same relative order (the *order property*). The key abstractions required to implement these properties are provided by *group communication systems*. They provide various 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).

Notably, the overlay protocols for reliable multicasting do not depend on any central server, so that there is no single point of failure. For this, *distributed agreement* protocols must be used, such as the Distributed Consensus protocol.

For the past 20+ years, many group communication systems have been implemented (see e.g. [4, 16, 5, 2, 11, 14, 6, 19, 15] among others). There are however few commercial systems (the examples are JGroups [15] and Spread Toolkit [19]). Unfortunately, various group communication systems usually have quite different application programming interfaces, which are non-standard, complex and language/platform dependent. Moreover, many of such systems are monolithic, i.e. it is not possible to replace their protocols or add new features. Using these systems to implement SOA services makes the code of services not easily reusable nor manageable (adaptable), which is a counterexample to the Service-

Oriented Architecture.

In this paper we propose an approach to designing an API of a group communication system, which is based on the *REpresentational State Transfer (REST)*. REST [9, 8] is a key design idiom in the Web services world that embraces a stateless client-server architecture, in which Web services are viewed as resources identified by their *Uniform Resource Identifiers (URIs)* [3]. Clients that want to request these services access their particular representation by transferring application content using a small globally defined set of methods. These methods describe an action to be performed on a given resource (consequently, by the corresponding service). Typically REST uses HTTP [7] and its methods GET, PUT, POST, and DELETE. This makes it easy to describe one RESTful Web service call to another Web service, e.g. a replicated service call to a group communication service. We need only supply a verb, a URI and (optionally) a few headers containing the message payload. Thus, REST gives us a uniform, simple way of using group communication systems by Web applications, fulfilling the SOA requirements, such as language/platform independence and easy software integration. Moreover, the use of HTTP usually enables us to communicate with servers behind fire-walls.

We had to solve some problems to realize this approach, mostly related to the REST characteristics and the constraints imposed on HTTP—a protocol which has not been originally designed for software communication. For example:

- the client would not be able to change state based on the responses of intermediary service calls; also,
- we had to provide means for the client to communicate both synchronously and asynchronously with the group communication service using purely the HTTP methods; and
- we needed to match error codes of the HTTP protocol to the incorrect behaviour of a group communication system.

Various authors pointed out significant limitations of the REST architecture style. For example, Khare and Taylor [12] discussed some of the limitations and proposed several extensions of REST (collectively called *ARRESTED*). They allow to model the properties required by distributed and decentralized systems. Similarly to them, we are not bound by the rules of the original model. REST cannot model group communication well. Therefore our goal was rather to design the RESTful interface to group communication, albeit sacrificing strict conformance to the original REST model. To emphasize that group communication cannot fully conform to REST, we say that our approach is “90% RESTful”. To illustrate our ideas, we have been developing *RESTGroups*—a group communication programming tool that can be used for developing resilient services on the Web. We think that the benefits of using our tool overcome the lack of REST purity. *RESTGroups* is an extension of Spread with a daemon and an API based on (some % of) REST over the standard HTTP [7]. RESTGroups functions as a front-end for Spread that is architecture- and language-independent, i.e. communicating services can be implemented with the use of a variety of programming languages and can run on different platforms. To our best knowledge, it is the first attempt to a RESTful group communication service; the distribution files and *javadoc* are available [1].

The structure of the paper is as follows. Firstly, we describe an architecture of our system and its main characteristics (e.g. statelessness). Then, we present an operation mode of the system using a small example. It demonstrates the use of the group communication programming interface based on REST/HTTP, and various communication protocols for interaction between the client and the server. The system has been verified by developing some use cases (or patterns); we sketch an example test application in the end. Finally, we conclude.

Group Communication – Functional Requirements

Group communication systems provide various primitives, e.g. for a unicast and broadcast within a group of processes, for maintaining group membership, for maintaining virtual (or view) synchrony, and for detecting process failures. Message broadcasting can be described with the use of two operations: a *broadcast* operation, which is used by a sender process to send a message to all processes in a group, and a *deliver* operation, which is used by a recipient process to deliver a broadcast message. Below we briefly describe functional requirements of a group communication toolkit. A more detailed specification of functional requirements can be found in many textbooks, for example [10].

Failure Models The design and implementation of group communication systems depend on a model of system failures. In the simplest case, we consider a *crash-stop* (or *crash-no-recovery*) model, in which a process that has failed stops its execution. In this model, a process is said to be *correct* if it does not crash; the notion of "correctness" refers to the whole execution time of a process. A process that crashes is said to be *incorrect* (or *faulty*). In this model the processes that crashed do not recover.

In a *crash-recovery* model, processes can be recovered after a failure. In this model, we consider a process to be incorrect if it crashes at least once and eventually cannot recover its state (*eventually down*), or if it crashes and recovers infinitely often (*unstable*). Otherwise, a process is correct, i.e. it never crashes (*always up*) or crashes at least once but eventually recovers and does not crash anymore (*eventually up*).

Broadcast Properties The simplest broadcast primitive is *Unreliable Broadcast*, which allows a message to be sent to all processes in a group, guaranteeing that if a message sender is correct, then all processes in the group will eventually deliver the message. However, if the sender crashes during the broadcast, then some processes in the group may not deliver the message. Obviously, this primitive is not much useful in systems, in which failures may occur. *(Regular) Reliable Broadcast* solves this problem; it guarantees the following properties:

- *Validity:* if a correct process broadcasts a message, then it eventually delivers the message;

- *Agreement:* if a correct process delivers a message, then all correct processes eventually deliver the message;
- *Uniform Integrity:* for any message, every process delivers the message at most once, and only if the message was previously broadcast.

Note that Reliable Broadcast allows executions, in which a faulty process delivers a message but no correct process delivers the message. *Uniform Reliable Broadcast* is a stronger version of Reliable Broadcast, which satisfies the Validity and Integrity properties defined above but replaces the Agreement property by the following:

- *Uniform agreement:* if a process (correct or not) delivers a message, then all correct processes will eventually deliver the message.

The Regular (or Uniform) Reliable Broadcast primitives provide a basis for stronger broadcast primitives, which have additional properties, e.g.:

- *FIFO order:* this property guarantees that messages sent by a process are delivered in the order of sending;
- *Causal order:* this property means that if some process has sent a message m_1 that caused sending of another message m_2, then each process in a group will deliver m_1 before m_2;
- *Total order:* this property means that messages sent by any different processes in a group will be delivered by all processes in the group in the same order (note that this property does not guarantee FIFO);
- *Generic order:* given a *conflict relation* $C : M \times M \rightarrow Boolean$, where M is a set of broadcast messages, if two messages are in conflict, they must be delivered to all processes in the same order. Otherwise, the messages are not ordered.

The (Uniform) Reliable Broadcast Protocols that support the above properties are called, correspondingly: *FIFO Broadcast*, *Causal Broadcast*, *Total-Order (or Atomic) Broadcast* and *Generic Broadcast.* Note that if the conflict relation C is empty, Generic Broadcast is reduced to Reliable Broadcast. Whereas, if all pairs of broadcast messages are in conflict, then Generic Broadcast is equivalent to Atomic Broadcast. RESTGroups however does not support Generic Broadcast since it is not available in Spread.

2 Architecture of RESTGroups

Spread Toolkit (or *Spread* in short) [19] provides a messaging service for point-to-point and multicast communication that is resilient to faults across local and wide area networks. Spread services range from reliable messaging to fully ordered messages with delivery guarantees. Spread supports several programming languages, for example C/C++, Java and Python.

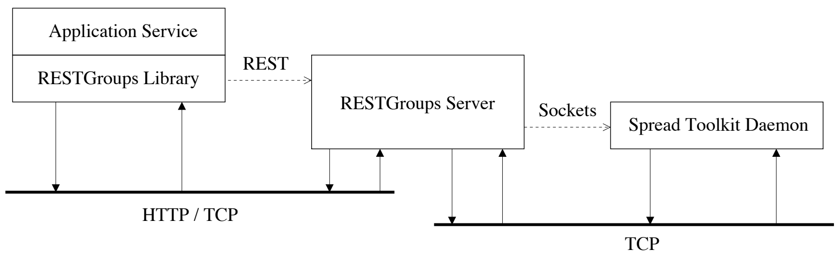

Figure 1: The RESTGroups system architecture

RESTGroups has been designed as the platform and language independent *front-end* for Spread. RESTGroups represents all Spread services as Web resources, accessible using the REST/HTTP architectural style. Since the Web is ubiquitous and the HTTP ports are usually not blocked, we can run the group communication service truly globally.

The RESTGroups system architecture is shown in Figure 1. It consists of a RESTful application interface to the Spread library that conforms to REST, and a daemon server for communication with a Spread daemon (they run on each computer that is part of the processor group). Below we sketch the interface (a more detailed description will be in the next section) and present the system's architecture. A complete description of RESTGroups, including a demo application, appeared in the technical report [13].

Programming interface RESTGroups provides a representation of Spread group communication services as resources, identified by URIs, with a simple but powerful API that only uses the following methods of the HTTP protocol for invoking the services:

- 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.

Figure 1 presents a fragment of a distributed system implementing replicated service (only a single replica is depicted). The RESTGroups system architecture shows that RESTGroups is an intermediary between a replicated *Application Service* and the Spread group communication system. Instead of communicating with a group communication system (GCS) directly, using its programming interface, a client of the RESTGroups system (the Application Service), uses exclusively the HTTP methods, following the REST/HTTP architectural style.

A suitable request of the HTTP protocol, possibly containing a XML document, is sent to a *RESTGroups server*, which is a front-end for the GCS. The GCS services are represented as Web resources, which are identified with the use of *Uniform Resource Identifiers (URIs)* [3]. The server translates client requests on these resources into concrete calls of the group communication services. In our case, these services are supplied by the *Spread Toolkit Daemon*.

System Deployment In the case of a concrete RESTGroups application, if the Application Service were replicated on n machines, then, in most cases, we would have: (1) n Spread Toolkit Daemons running on any n machines, and (2) from 1 to n RESTGroups Servers that are communicating with them, usually deployed on the same machines as Spread daemons.

The clients of the replicated Application Service (not shown in Figure 1) can communicate with any replica of the service. Each service replica (Application Service) connects to a dedicated RESTGroups server. Each of the RESTGroups servers can interact with any Spread daemon, using the sockets on TCP. Except when broadcasting of messages can be optimized at the low level of network protocols (which is possible in the local-area networks) the Spread Toolkit Daemons communicate using the *IP unicast* protocol.

Statelessness The RESTGroups server does 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; the details of this mechanism will be explained in the following section.

Importantly, the RESTGroups server does not have any representation in the group communication system, which is the *back-end* of RESTGroups. In particular, the unique client identifiers, generated by the GCS, are used exclusively by the GCS, independently from the RESTGroups. 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 RESTGroups clients can establish connections with separate RESTGroups servers, that are available within the same group.

3 API and Communication

Group communication systems, such as Spread Toolkit, provide various primitives (or services), e.g. for a unicast and broadcast within a group of processes, for maintaining group membership, for maintaining virtual (or view) synchrony, and for detecting process failures. In RESTGroups all these services are represented as resources, maintained by RESTGroups servers.

In this Section, we describe our API and discuss the communication of a client `userA` with a RESTGroups server. Let us assume that the RESTGroups server is available at the address: `http://mydomain.com:8182`. The following components are required for the communication with the server:

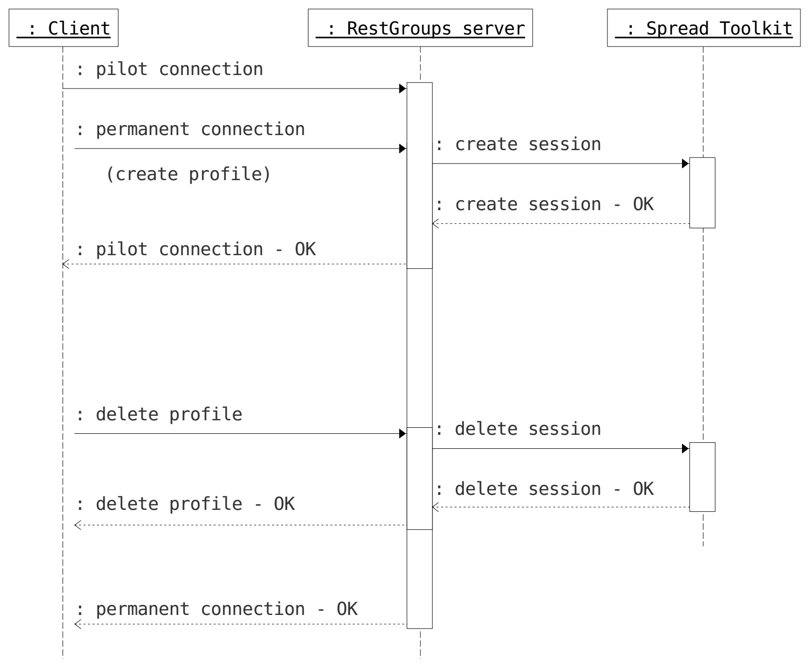

Figure 2: Successful connecting and disconnecting from the RESTGroups server

- a file transfer library supporting HTTP (e.g. libcurl or restlet),
- a library for building and parsing XML documents (e.g. jdom).

3.1 Connecting to the RESTGroups Server

HTTP is a stateless protocol for the client-server communication. In order to execute a given action by a server, a client initializes connection with the server and sends a request to it. The request contains all the information that are needed by the server to process the action. After processing the action, the server sends back a response message and the connection is closed. Therefore, using HTTP as a transport protocol in the group communication system does not seem natural. A permanent connection would be more useful, since it can allow the system to detect client's failure when the connection is broken.

Therefore, the connection with the RESTGroups server is accomplished using two requests 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. In Figure 2, we illustrate making a successful connection and disconnection with the RESTGroups server.

Connection with the RESTGroups server is identified by a unique identifier `pilotConnectionToken`, created with the use of UUID numbers [20]. 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.

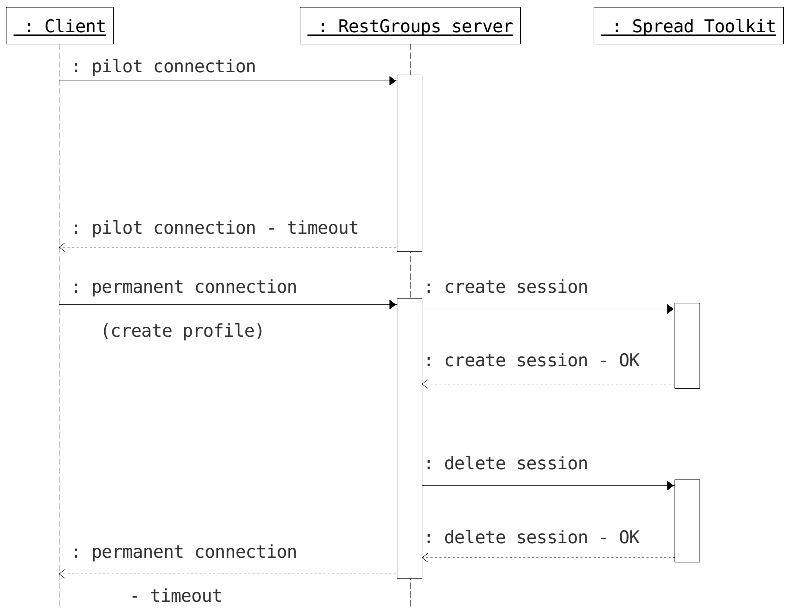

Figure 3: Unsuccessful session creation due to a connection timeout

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 (see Figure 3),
  - `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 (see Figure 3),
  - `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 (see Figure 4).

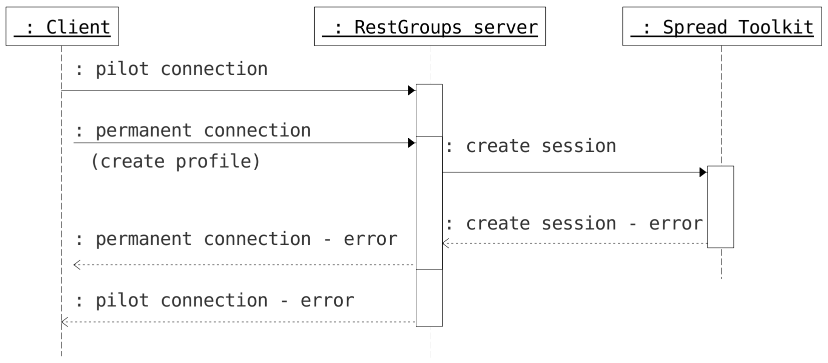

Figure 4: Unsuccessful session creation due to errors returned by the group communication system working as *back-end*

3.2 Disconnecting 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.

3.3 Joining 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,

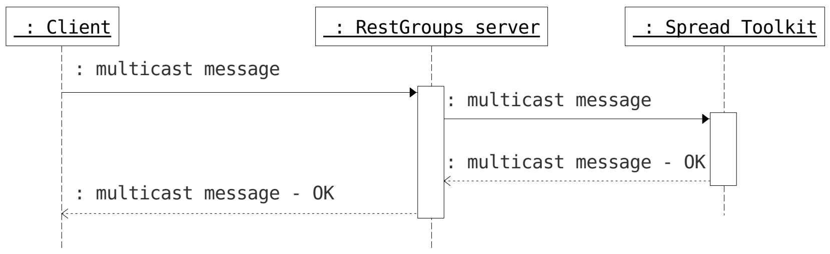

Figure 5: Sending a message

- `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.

The process of joining a group can be described using a sequence diagram in Figure 5, which illustrates sending of a message.

3.4 Leaving 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).

A sequence diagram for leaving a group is similar to the diagram illustrating sending of a message (see Figure 5).

3.5 Sending 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/client-MessageSingleGroup.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.

3.6 Checking 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.

3.7 Non-Blocking Reception of Messages

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 our system [1] and in the source code of `RESTGroups-Client` in the `restgroupsClient` package.

3.8 Blocking Reception of Messages

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 (see Figure 6). 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.

4 Test Application

To verify usefulness of our approach, we have designed a small distributed test application. It implements a text console, analogous to the *User* application included in Spread release. The console executed on a given machine allows the clients to create new custom groups and the users which can dynamically join/leave custom groups and send/broadcast messages with specific guarantees. Messages can be received using both the blocking and non-blocking modes. The test application is included in RESTGroups release [1] and described in the technical report [13].

In the future, we plan to use RESTGroups in the implementation of a system for managing and monitoring RESTful Web services (ongoing work).

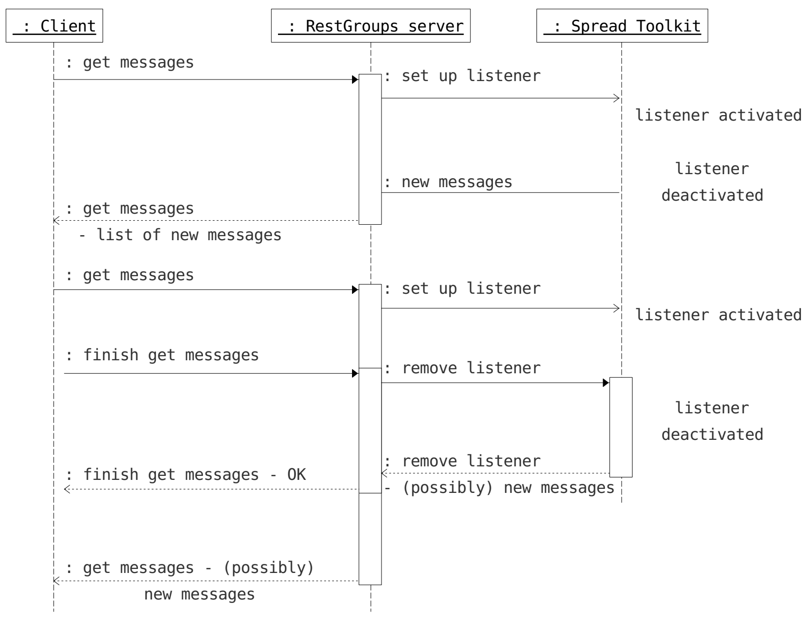

Figure 6: Blocking reception of messages

5 Conclusion

RESTGroups wraps functionality of a chosen group communication system and exposes it through a uniform interface that is partially consistent with the RESTful approach. However, due to the nature of the problem, some REST principles cannot be captured, such as stateless interaction with certain group communication resources.

The usage of the open and popular hypertext transport protocol (HTTP) and the open XML data format, which are both independent of the programming language and the platform, makes building of distributed applications that require group communication abstractions easier and time/cost-effective. This claim requires more evaluation which we leave for the future work.

Acknowledgments. This work has been partially supported by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03.01-00-008/08.

References

[1] (2010). *RESTGroups*. http://www.cs.put.poznan.pl/pawelw/restgroups/.

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

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

[4] Ken P. Birman & Robbert Van Renesse (eds.) (1994): *Reliable distributed computing with the Isis toolkit*. IEEE Computer Society Press.

[5] Danny Dolev & Dalia Malki (1996): *The Transis approach to high availability cluster communication*. *Communications of the ACM* 39(4), pp. 64–70.

[6] EPFL (2006). *Fortika*. http://lsrwww.epfl.ch/crystall/.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach & T. Berners-Lee (1999): *Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force. RFC 2616 (Draft Standard). Updated by RFC 2817.

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

[9] 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.

[10] Rachid Guerraoui & Luís Rodrigues (2006): *Introduction to Reliable Distributed Programming*. Springer.

[11] Mark Hayden (1998): *The Ensemble System*. Technical Report TR98-1662, Department of Computer Science, Cornell University.

[12] Rohit Khare & Richard N. Taylor (2004): *Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems*. In: *Proceedings of ICSE '04: the 26th International Conference on Software Engineering*, pp. 428–437.

[13] Tadeusz Kobus & Paweł 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 Poznańska.

[14] Hugo Miranda, Alexandre Pinto & Luís Rodrigues (2001): *Appia, a flexible protocol kernel supporting multiple coordinated channels*. In: *Proc. of ICDCS '01*.

[15] Red Hat (2009). *The JGroups Toolkit*. http://www.jgroups.org/.

[16] Robbert Van Renesse, Kenneth P. Birman & Silvano Maffeis (1996): *Horus: A flexible group communication system*. *Communications of the ACM* 39(4), pp. 76–83.

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

[18] 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.

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

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