RESTGroups for Resilient Web Services

Tadeusz Kobus and Paweł T. Wojciechowski

Poznań University of Technology
Institute of Computing Science
60-965 Poznań, Poland
{Tadeusz.Kobus,Pawel.T.Wojciechowski}@cs.put.poznan.pl

Abstract. Service resilience, defined as the continued availability of a service despite failures and other negative changes in its environment, is vital in many systems. It is typically achieved by state machine replication using group communication middleware for coordination of service replicas. In this paper we focus on systems that represent critical data as Web resources identified by Uniform Resource Identifiers (URIs). The best examples of such systems are RESTful web services. We present *RESTGroups*: a group communication *front-end* for RESTful web services. Our system is based on Spread – a popular group communication toolkit. Contrary to Spread and other such toolkits, we represent group communication services as resources on the Web, addressed by URIs. In the paper, we describe the system architecture and the API.

1 Introduction

The *Web* can provide a common, language-independent platform for interoperable services that work together to create seamless and robust systems. However, we must ensure that each individual service is *resilient*, i.e. it is able to withstand unpredictable and difficult conditions, such as sudden and significant degradation of network latency or failure of dependant services. In our work, we focus on RESTful web services. *REpresentational State Transfer (REST)* [6, 5] embraces a stateless client-server architecture, in which web services are viewed as resources identified by URIs. Clients that want to request these services access their particular representation by transferring application content using a small globally defined set of methods. The methods describe an action to be performed on a given resource (consequently, by the corresponding service). Typically REST uses HTTP [4] and its methods GET, PUT, POST, and DELETE.

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* [19]. In this approach, each non-fault replica receives every request (the *agreement property*), and each non-fault replica processes the requests in the same relative order (the *order property*). The key abstractions required to obtain these two essential properties are offered by *group communication systems*. They provide reliable multicast transport protocols with a range of delivery options, e.g. causally- , fifo- and totally-ordered multicasts in a group of processes. The protocols are fully distributed, i.e. they do not depend on any central server and so there is no single point of failure. For the past 20+ years, many group communication systems have been implemented (e.g. JGroups [17] and Spread [20]; see also [11] for other references). Unfortunately, they have quite different APIs, which are language dependent and complex. Moreover, many of group communication systems are monolithic, so it is not possible to easily replace their protocols or add new features. Using these systems to implement resilient web services makes the code of the services neither easily reusable nor interoperable with other services, which is a counterexample to the openness of the Web. Moreover, none of the system that we know offers a REST/HTTP-based interface.

In [16], the authors discuss some examples of service replication middleware systems developed at the academia (e.g. WS-Replication [18]); however, none of these systems supports RESTful web services. The industry solutions for web service resilience are usually based on the SOAP-based WS-* standards that were not designed for service replication. The concrete implementations of service resilience are usually built on queueing or publish-subscribe systems. This approach does not benefit from the group communication protocols that have been optimized for state machine replication. When it comes to RESTful approaches to web service implementation, group communication solutions that offer all associated properties and guarantees are unknown to us. At the moment there is a lack of standards in the domain of group communication intended for the REST style. This provided motivation for our work described in this paper.

We introduce a group communication *front-end* for replication of RESTful web services – *RESTGroups*. A brief announcement of our work appeared in [8]. In this paper, we describe the system architecture and the API. Our system is based on Spread [20, 2] – a popular group communication toolkit implementing protocols for reliable, ordered multicasts and group membership. Contrary to Spread and other such toolkits, we represent group communication services as resources on the Web, addressed by URIs. Thus, RESTful web services and their clients can use group communication services in the same style as they communicate among themselves. Moreover, since firewalls usually do not block the HTTP protocol, RESTGroups supports communication across firewalls. The system has been implemented and the distribution files are available [1].

2 RESTGroups Design and Service Replication

RESTGroups is a group communication *front-end* for RESTful web services. Our current implementation is based on Spread Toolkit. Spread [20, 2] is a monolithic group communication system, consisting of a daemon program, client libraries, and a system monitor. RESTGroups represents group communication services provided by Spread, as resources on the Web, addressed by URIs. Spread's API consists of many functions with bindings available for several programming languages: C/C++, Java, Perl, Python, and Ruby. RESTGroups has a small but powerful API that consists of just four methods of the HTTP protocol: GET, POST, PUT, and DELETE. They can be used 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, and reliable message multicasting with a wide range of guarantees concerning delivery of messages to group members (e.g. causally-, fifo- and totally-ordered delivery).

2.1 System design

A system built using RESTGroups consists of four types of communicating components: Web Service, Client, RESTGroups Server (RESTGr Server in short), and spreadd (which is a daemon of Spread Toolkit). Web Service is a user-defined RESTful web service. The RESTGr Server acts as a *proxy* between Web Service and group communication protocols that are implemented by spreadd. The communication between Client and Web Service, as well as between Web Service and RESTGr Server uses the REST/HTTP style. RESTGr Server and spreadd communicate using TCP and may or may not run on the same machine.

Group communication services (provided by Spread) are represented as Web resources identified by URIs. Instead of calling Spread methods, a user-defined Web Service invokes only four methods of the HTTP protocol (i.e. GET, PUT, POST or DELETE). Then, a suitable HTTP request, possibly containing an XML document, is sent to RESTGr Server that translates it into a group communication call to spreadd. A crash of RESTGr Server results in the disconnection of all Web Services that are using this server. They can establish connection with another RESTGr Server which is available within the same group. Next, we present an architecture of a system, in which the RESTGroups components will be replicated for resilience.

2.2 Service replication

In Figure 1, we show an architecture of an example system in which RESTGroups has been used for replication of a RESTful web service. There are three service replicas (each one called *Web Service replica*) perceived by the clients as a single web service, represented as a large circle. Clients can issue REST/HTTP requests to any of them. Each Web Service replica connects to two RESTGr Servers using HTTP, so it can tolerate a crash of one server. Each RESTGr Server has its own spreadd so that partial failure of the Spread group communication system used as the back-end is also tolerated.

In general, replicating a web service to tolerate at most $\lfloor (n-1)/2 \rfloor$ machine crashes, requires the following steps:

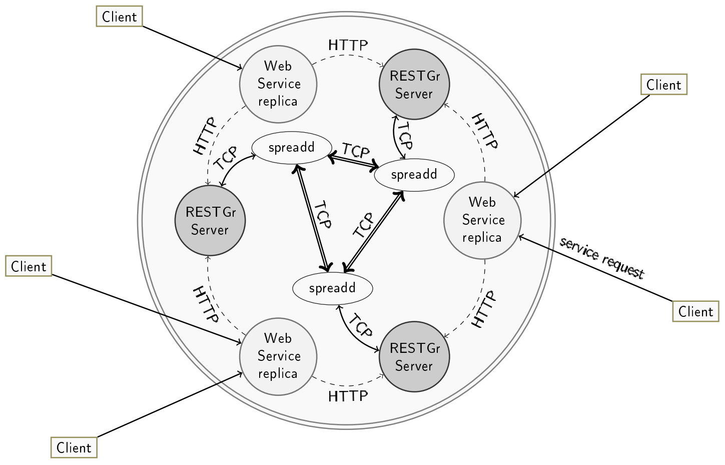

Fig. 1. Replication of a RESTful web service

- spawning n Spread daemons (spreadd) on n independent machines;
- spawning n RESTGroups servers on different machines; each server communicates only with one Spread daemon (usually located on the same machine);
- spawning n instances of the RESTful web service on different machines (in this case, they would run on the same machines as Spread daemons); each service replica can communicate with one or many RESTGroups servers.

The service developers can use the replicated state machine approach [19, 22] to implement a resilient RESTful web service, as follows. After system start up, a group is created to which all Web Service replicas must join. A client can issue a request to any known replica which then forwards the request to the RESTGr Server that is alive; the latter broadcasts the request in the group. All client requests issued to (any replica of) the web service are delivered within the group totally ordered. Thus the requests will be processed by each replica in exactly the same order; the client will obtain only one reply to each request. We require the web service to be deterministic, so that all replicas will make transition to the same state in response to the same sequence of requests issued by clients. In the case of a replica crash, the clients may have to repeat its request to another Web Service replica after a timeout.

2.3 Statelessness

RESTGr Servers are almost *stateless* – they only store data that are necessary to maintain group communication sessions for the connected Web Service replicas. Moreover, RESTGr Server does not have any representation in the group communication system that is the back-end of RESTGroups. However, unique client IDs generated by Spread are used by RESTGroups.

Various authors pointed out limitations of the REST architectural style. For example, Khare and Taylor [7] discussed some of the limitations and proposed 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 since REST cannot model group communication well (as the RESTGr Server is not 100% stateless). Therefore our goal was rather to design the REST-inspired interface to group communication, albeit sacrificing strict conformance to the original REST model.

3 RESTGroups API Calls

In this section, we explain the calls of RESTGroups API using a few simple examples. A complete description of the API is in the User Guide, available electronically [1]. The following methods of the HTTP protocol are used, where resources represent some group communication services or data structures (such as a mailbox):

- GET is used to perform a query on a resource, e.g. to retrieve messages from the mailbox (in a blocking or non-blocking manner);
- PUT is used to create a new resource, e.g. to extend a process group with a new process; the server responds with a status indicating success or failure;
- POST is used to update existing resources, e.g. to connect to the server on system start-up (this operation is executed only once) or to send/broadcast a new message;
- DELETE is used to remove a resource, e.g. to remove a process from a process group; in some cases, the update and delete actions may be performed with POST operations as well.

Consider the RESTGroups server located at http://mydomain.com:8182 and a RESTGroups client (or client, in short), denoted userA, located at some other site. For example, userA could be the Web Service replica in Figure 1. Below we describe the following operations: connecting to the server, sending messages, and message retrieval.

3.1 Connecting to the server

HTTP is a stateless protocol for 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

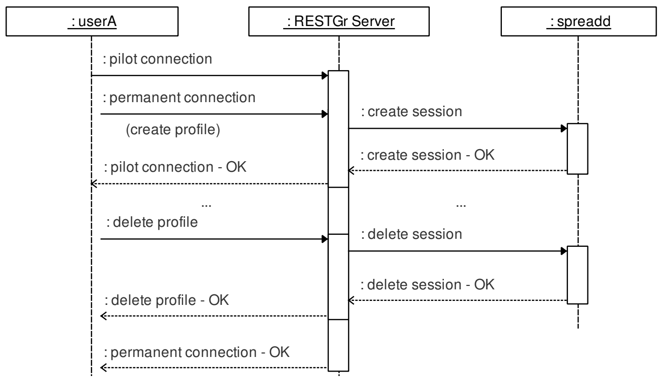

Fig. 2. Successfully connecting and disconnecting from the RESTGroups server

used to ask the server to set up a resource which represents a new communication session. The session is created using the second request, called the *permanent* request. The server does not respond to this request, 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 RESTGr Server is identified by a unique identifier pilotConnectionToken, created with the use of random UUID numbers [21]. The UUID number created by a client is sent in the XML format in the bodies of both the pilot and 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 a 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>
```

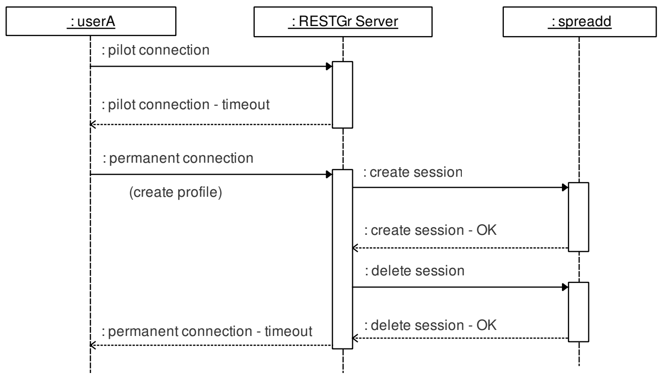

Fig. 3. Unsuccessful session creation due to a connection timeout

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 clients, and 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 fails, suitable error messages are received, e.g. in response to the pilot or permanent request, HTTP's 408 'Request Timeout' error can be received if one of the two requests has not been received in a predefined period of time (see Figure 3).

3.2 Sending messages

There are two possible ways of sending messages to a group of users or to a single user, identified by the URI of the private group to which it belongs (only one user can belong to a given private group). The first way (see Figure 4) 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. When using the second way, 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

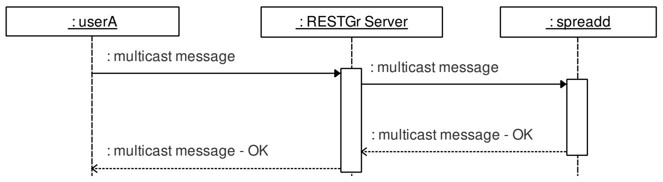

Fig. 4. Sending a message

an individual user, must be represented by a resource, identified by URI. This approach is convenient if messages are addressed to a single user only.

Consider a user-defined group named customGroup. Sending a message to this group 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 the proper XML schema. The following sections (or tags) of the structure must be defined: guarantee – the reliability and ordering guarantees of message delivery, type – a message type, groups – a list of addresses, and finally data – the message payload.

The following guarantees of message delivery are supported:

– 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 appear as 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 earlier.

Upon successful message sending, the RESTGr Server returns a response message with the 204 'Success No Content' status code. In the case of an error, the server returns the HTTP error message, e.g. 400 'Client Bad Request' – if the client with the sessionID identifier in the request's 'cookie' does not have an active RESTGroups session, or 503 'Service Unavailable' – if an error occurs during the disconnection from the group communication system.

3.3 Reception of messages

The RESTGroups system offers two types of mechanisms for reception of messages: *blocking* (synchronous) and *non-blocking* (asynchronous). A user can also check if there are any unread messages waiting on the RESTGroups server without fetching them. Below we describe the blocking reception mechanism; the non-blocking reception mechanism has a similar syntax.

Performing the following GET request by a client is suspended until a new message (or messages) will be received by the client:

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

A response to this request is an XML document which contains aggregated messages that have been sent (or broadcast) to the client; each message contains the names of the broadcast group and of the message sender. Messages are sent to a client as soon as they arrive to the RESTGroups server. The structure of responses in the case of non-blocking messages is similar, except that the "no messages" response can also be returned.

In order to stop receiving messages, the client should issue the DELETE request:

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

4 Related Work

In this section, we describe related work on group communication support for web services. We begin from discussing the industry standards, followed by example research projects. This work is for SOAP-based web services only; we are not aware of similar work done for RESTful web services.

In SOAP-based web services, distributed processes communicate messages, typically wrapped in the XML format, using the *Simple Object Access Protocol (SOAP)*. Essentially, SOAP means sending remote procedure calls (RPC) through standard HTTP ports, using an XML envelope. REST emphasizes the element of using standardized URIs, and also giving importance to the HTTP verb used (i.e. GET, POST, PUT, or DELETE). Benefit of the RESTful interface is that requests and responses can be short – in contrary to SOAP that requires an XML wrapper around *every* request and response. On the other hand, SOAP can easier transport any attached files and has better tool support. Since group communication protocols exchange many control messages, shorter processing time of messages in REST means better performance of these protocols. However, we do not present evaluation results since our contribution is mainly the design of the group communication interface for RESTful web services, not a group communication system. In particular, we might use some other back-end system instead of Spread and obtain different performance.

WS-ReliableMessaging [15] is an OASIS standard describing the protocol for reliable unicast-only message communication using SOAP. The standard does not specify multicast (or broadcast) communication. It defines two components: *Remote Messaging Source (RMS)* on the sender side and *Remote Messaging Destination (RMD)* on the receiver side. These two components communicate using SOAP-messages. For this, a communication link is created between RMS and RMD, and all messages exchanged using this link are given a sequence number. The programmer can choose among the following levels of *delivery assurances*, e.g. at-least-once, at-most-once, exactly-once and in-order. The above properties can provide building blocks for implementing group communication abstractions above the WS-ReliableMessaging. In RESTGroups, the unicast communication with analogous guarantees can be achieved by defining a group to which only the sender and the receiver belong, and using a POST method with the required semantics (unreliable, reliable, or fifo).

WS-BaseNotification [13] and *WS-BrokeredNotification* [14] are OASIS standards describing the protocols for one-to-many communication of SOAP messages. The communication is based on the *publish-subscribe* model. The system users can create *topics* of messages, to which the message recipients (or *consumers*) can subscribe. The standard allows to have a separate *subscriber* that subscribes a number of consumers to a given topic. When a message sender (or a *publisher*) publishes a message on a given topic, the message is propagated to all consumers who subscribed (or have been subscribed) for that topic and their subscription remains active. The *WS-BrokeredNotification* standard also introduces a *broker*, who is responsible for recording published messages of a given topic, and resending them to all *consumers* that have subscribed to that topic.

However, the WS-BaseNotification and WS-BrokeredNotification standards focus on the information exchange protocol only, leaving the issues of reliable communication to “a delivery mechanism for transmission”, where transmission properties are unspecified: “depending on the actual delivery mechanism, this transmission might be reliable or might be done on a best-effort basis” [13].

In commercial applications, *message queueing* systems are often used as the mechanism for reliable message transmission, including the one-to-many interaction. They provide an asynchronous communication protocol between distributed, loosely coupled processes. The sender and a receiver of a message do not need to be accessible at the same time, for the message to be delivered (by default, in the FIFO order). When a receiver of a message is not accessible, the message will be stored in a queue until it can be delivered. This property is called *durability*. Many implementations of queueing systems support resilience to system failures. This is usually done by implementing a message property called *persistence*. After receiving a persistent message, a queueing system sends a message receipt acknowledgment but only after the message had been stored in non-volatile memory. The persistence property guarantees that a message characterized by this property will never be lost, even in the case of runtime failures. Many queueing systems have been developed for the last few decades. This resulted in a variety of protocols and APIs (including REST/HTTP). However, many consider the *Java Message Service (JMS)* [12] to be a *de facto* standard of a queueing system. JMS assumes both the one-to-one and one-to-many mode of communication, where the latter uses the publish-subscribe model. In [10], the authors show that it is possible to build a group communication system based on JMS, offering a notion of a group, a membership service and a total-order broadcast. However, such approach adds additional layers of abstraction when compared to RESTGroups. On the contrary, the group communication protocols designed for the replicated state machine are more efficient.

The closest work to ours is *WS-Multicast* [18] that has been designed as a broadcasting service for SOAP-based web services. The service is built on the JGroups group communication system [17]. It uses its own transport layer module for message communication based on SOAP. A WSDL interface has been defined, making WS-Multicast a web service itself. Since WS-Multicast only replaces the transport layer of the JGroups system, leaving the rest of the protocol stack unchanged, all the assurances offered by JGroups remain in place. Nonetheless, as was noted, the use of SOAP involves sizable cost stemming from the character of this protocol. Thus, in the final version of the proposed service, parsing XML data was being avoided.

5 Conclusion

In this paper, we demonstrated that group communication middleware, such as Spread, can be easily extended to support RESTful web services. RESTGroups wraps functionality of group communication middleware and exposes it through a uniform interface based on the HTTP protocol. We have discussed an example application of our system – replication of RESTful web services. We also emphasized that systems like RESTGroups cannot be 100% RESTful since some REST principles, such as client-server stateless interaction, cannot be captured in this type of application.

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. RESTGroups. http://www.it-soa.pl/restgroups, 2010–2011.
2. Yair Amir and Jonathan Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, Dep. of CS, Johns Hopkins Univ., 1998.
3. T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. Internet Engineering Task Force, August 1998. RFC 2396.
4. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, June 1999. RFC 2616 (Draft Standard). Updated by RFC 2817.
5. Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
6. Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM TOIT*, 2(2):115–150, 2002.
7. Rohit Khare and Richard N. Taylor. Extending the Representational State Transfer (REST) architectural style for decentralized systems. In *Proc. ICSE '04*, May 2004.
8. Tadeusz Kobus and Paweł T. Wojciechowski. A 90% RESTful group communication service. In *Proc. of the DCDP Workshop '10*, June 2010. Available electronically at http://arxiv.org/html/1006.1689v1.
9. Tadeusz Kobus and Paweł T. Wojciechowski. A 90% RESTful group communication service. Technical Report RA-02/10, Institute of Computing Science, Poznań University of Technology, May 2010.
10. Arnas Kupšys, Stefan Pleisch, André Schiper, and Matthias Wiesmann. Towards JMS compliant group communication - A semantic mapping. In *Proc. of NCA '04*, August 2004.
11. Sergio Mena, André Schiper, and Paweł T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. of Middleware '03: the 4th ACM/IFIP/USENIX Middleware Conference*, LNCS 2672, June 2003.
12. Sun Microsys. Java Message Service. http://java.sun.com/products/jms/, 2009.
13. OASIS. *Web Services Base Notification 1.3*, 2006.
14. OASIS. *Web Services Brokered Notification 1.3*, 2006.
15. OASIS. *Web Services Reliable Messaging 1.1*, 2007.
16. Johannes Osrael, Lorenz Froihofer, and Karl M. Goeschka. What service replication middleware can learn from object replication middleware. In *Proceedings of MW4SOC: the 1st Workshop on Middleware for Service Oriented Systems*, December 2006.
17. Red Hat. The JGroups toolkit. http://www.jgroups.org/, 2009.
18. Jorge Salas, Francisco Pérez-Sorrosal, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. WS-Replication: A framework for highly available Web services. In *Proc. of WWW '06*, May 2006.

19. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM CSUR*, 22(4):299–319, 1990.
20. Spread Concepts LLC. The Spread toolkit. http://www.spread.org/, 2006.
21. The Internet Society. A Universally Unique IDentifier (UUID) URN Namespace. http://www.ietf.org/rfc/rfc4122.txt, 2005.
22. Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustawo Alonso. Understanding replication in databases and distributed systems. In *Proceedings of ICDCS '00: the 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474, April 2000.