On Actors and the REST

Janne Kuuskeri and Tuomas Turto

Department of Software Systems
Tampere University of Technology
PL 553, 33101 Tampere, Finland
{janne.kuuskeri,tuomas.turto}@tut.fi

Abstract. The prevalence of RESTful services requires that we pay closer attention to how the principles that underlay REST are realized in actual services being implemented. This is especially crucial as REST is being applied to problem domains that require complex operations such as transactions. In this paper we investigate the relationship between RESTful web services and the actor model of computation. We suggest that by formulating RESTful services as a network of actors we can achieve deeper understanding what it means for a service to be RESTful.

1 Introduction

In his thesis Fielding [9] discusses how the Web's architecture as a distributed hypermedia system evolved in its early stages. In particular, the thesis describes how the Representational State Transfer, or REST, architectural style was used to guide the development. Given that REST pinpoints the architectural constraints that have made the Web a success, it has been argued that also the web services should be architected in a similar fashion [16]. These so-called RESTful services would then embrace the way applications on the Web are intended to function.

Often, however, the well-intended discussion on the principles of RESTful web services degenerates into heated debate about the merits of REST compared to the big web services implemented using the WS-* stack [20]. Although the architectural choice between a RESTful approach and the WS-* stack does require significant consideration and there are arguably better problem domains for each of them [14], this sort of comparative argumentation does not deepen our knowledge of RESTful services as much as it could.

Although the key principles of REST, such as the use of resources and the hypertext as an engine of application state in the sense of [9], are widely agreed upon, there seems to be conceptual confusion regarding how they do manifest and how they should manifest themselves in actual RESTful services. Discussion about the fundamentals of REST and a deeper understanding of these key issues is urgently required as RESTful services embed more complex behavior such as transactions [15]. Furthermore, as REST itself is being extended [7], we need solid understanding on how to apply the ideas behind REST.

In this paper we investigate the principles of REST in the framework of the actor model of computation [3]. The actor model has previously been used to

B. Benatallah et al. (Eds.): ICWE 2010, LNCS 6189, pp. 144–157, 2010.

reason about distributed systems in general [1] and as the basis for Internet-wide middleware development [2]. Also, from a more pragmatic point of view, programming languages based on the actor principles, such as Erlang [4], have received acclaim for their ability to implement RESTful services in a natural style. We conjecture that it is indeed the actor model that makes them especially suitable for implementing RESTful services.

The main contribution of this paper is an explicit investigation into the relationship between the actor computation model and the principles of REST. We show how a restricted actor model can be used to understand the principles underlying the RESTful paradigm and especially how an actor system embodies the idea of hypertext as an engine of application state. Moreover, we suggest a notation for describing RESTful systems.

The rest of the paper is structured as follows. In Section 2 we review the key ideas behind RESTful services and introduce the actor model of computation. Next, in Section 3, the underlying principles of REST and the actor model are related to each other. In Section 4 we introduce a notation for expressing RESTful services in a restricted form of actors and apply the result to an example. Section 5 discusses the pros and cons of the suggested approach and Section 6 provides a review of related work. Section 7 concludes the paper with some final remarks.

2 Background

In this section we introduce REST and the actor model of computation separately in order to provide the necessary background for the rest of the article.

2.1 RESTful Architectural Style

Although REST is a general architectural style for building network-based software, in this article we consider RESTful interfaces only in the context of web services. In this resource oriented architecture, resources are exposed by the servers and consumed by the clients using HTTP methods [8]. A resource is accessed via a URL and its state is transferred using its representation. A key characteristic of a RESTful interface is the clear division of application state between the client and the server. In the following we inspect these essential features of REST in more detail.

Resources and URLs. In a RESTful interface, everything is a resource. Conversely, everything that can be represented by a URL can be a resource. A resource can be static (e.g. `/blog/2010-01-03`) or its representation can change over time (e.g. `/blog/latest`). Moreover, the resource can also represent a list of things (e.g. `/blog/2010/`). Although every resource must be addressable by a URL, the REST itself does not mandate any scheme for constructing URLs. Human readable URLs are preferred, as they should suggest how to use the interface, but not required by REST. However, the URL should not contain any information about any operation that might be applied to the resource (e.g. `/add/` or `/remove/`).

Connectedness. A property that is closely related to addressability is connectedness. For a client to be able to consume a RESTful service, it needs to know the addresses of all the necessary resources. These resource identifiers can be pre-configured into the client but this is not desirable because it does not enforce connectedness. Instead, the server should guide the client by letting it know about all meaningful resources and then the client can make the decision which path to choose. The server can do this be sending hypermedia links to other resources it exposes. This pattern is also known as Hypermedia as the Engine of Application State.

Methods of REST. For RESTful services, the most commonly used HTTP methods are `POST`, `GET`, `PUT` and `DELETE`. These methods define the basic CRUD[1] operations for resources. These methods can be categorized in accordance to how they operate on resources. The `GET` method is said to be *safe* because it has read only semantics on the resource, implying that it is meant only for information and data retrieval without any side effects. With `PUT` and `DELETE`, `GET` is also *idempotent*, since it does not matter whether the operation is applied once or several times on a resource. The end result is always the same. For instance, it does not make a difference if a resource `/blog/2010-02-09` is deleted once or twice; it will not exist afterwards.

Statelessness. Honoring the principles of HTTP, REST is strictly stateless. Any application state is stored only on the client. The server, on the other hand, only stores the resources and their states. This also means that each request to a resource must be self-containing; i.e. each request must contain all the information needed to carry out the request. So, the server does not need to – nor it should – know anything about any previous or future request made by the client. Each request should happen in complete isolation from any other request.

Uniform Interface. A very unique characteristic of REST is the uniform interface that it imposes on the services that employ it. Where RPC style services expose bespoke objects and methods for clients, RESTful interfaces expose resources with a fixed set of HTTP methods and return values for each resource. Most importantly this is a profound change in the way the interface for a web service is designed. Everything needs to be designed in terms of resources as opposed to functionality.

2.2 The Actor Model of Computation

Hewitt et al. [11] originally suggested the actor model of computation in the context of enhancing programming languages for artificial intelligence. Our exposition follows [3] that describes actors as a model for concurrent computation in distributed systems.

[1] Create, Read, Update, Delete.

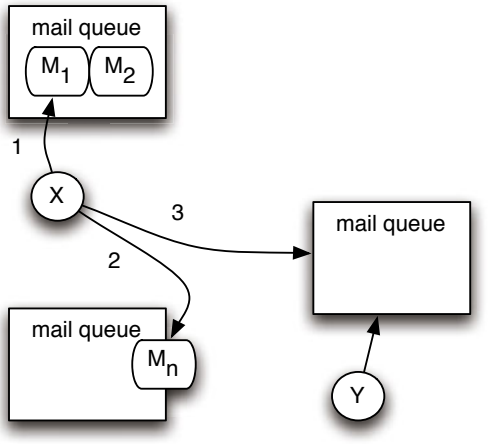

Fig. 1. Sending messages and creating actors

Actors. In the actor model of computation a system is composed of a multitude of computational agents. Each of these computational agents is an actor. An actor is an active self-contained entity that is identified by its address. Actors encapsulate all the necessary information that is required, as specified by the actor's behavior, for it to function as a part of the system. Actors do not share any state and communicate only by passing messages.

For each address there exists a corresponding conceptual mailbox that queues the incoming messages. In the actor model message passing guarantees that messages sent to an actor are eventually delivered, but it does not guarantee the time it takes to deliver a message, nor does it specify the order in which sent messages arrive at the target actor's mail queue. Furthermore, it is not possible to send a message to an arbitrary actor without first knowing its address.

Operational behavior. The driving force in an actor system is the process of sending and receiving messages. In fact, it is the only way the computation in an actor system makes progress. Each time an actor receives a message, it

- must decide on its replacement behavior.
- can create new actors.
- can send messages to other actors.

These three actions can occur concurrently. Once the replacement behavior has been decided another instance of an actor machine is created to represent the actor. This new actor machine is then able to process the next message from the actor's mail queue.

This process is shown in Figures 1 and 2[2]. In Figure 1 an actor with two messages in its message queue is shown. The actor is represented by the actor machine X that processes the first message in the mail queue, M_1 (1). As a part

[2] Figures adapted and extended from Figure 3.2 in [3].

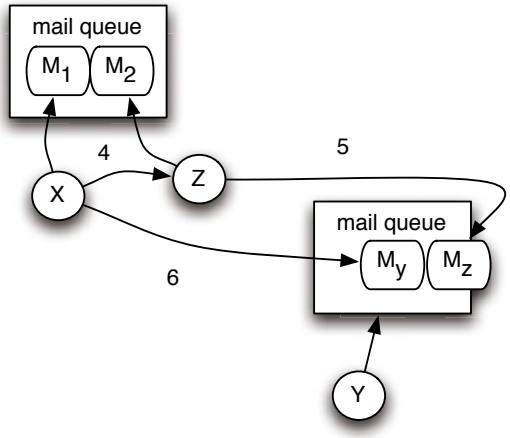

Fig. 2. Concurrent actor machines

of its behavior it can send messages to other actors (2) and create new actors (3). Once the actor machine X has done enough work to decide its replacement behavior with respect to the message it received and its environment, another actor machine Z that corresponds to this replacement behavior is created (4). This is depicted in Figure 2. It is important to note that once the replacement behavior has been decided, both actor machines run concurrently. That is, the new actor machine Z can create new actors or send messages to existing actors (5), while X is free to do the same (6).

3 Relating Actors and REST

In this section we relate the essential concepts of the actor model to those of REST and investigate the relationship. Although it might seem, as actors represent a model of computation and REST is an architectural style, that the connection between them is far-fetched, this is not the case. Let us consider the following rule of thumb given in the literature. Hewitt [10] defines the actor programming methodology to consist of

1. Deciding on the natural kinds of actors (objects) to have in the system to be constructed.
2. Deciding for each kind of actor what kind of messages it should receive.
3. Deciding for each kind of actor what it should do when it receives each kind of message.

On the other hand, adapting from Richardson and Ruby [16], for RESTful web services we get the following

1. Figure out the data set and split it into resources.
2. Name the resources with URLs and expose a subset of the uniform interface.

3. Design the representation(s) for the resources.
4. Integrate the resources with one another using hypermedia links.

Both approaches consider self-standing individual entities as the basic elements: actors in the actor model of computation and resources in REST. These elements are then identified by unique addresses, and they communicate by passing messages. In this section we first elaborate the relationship and then present an example where a RESTful service is represented in terms of the actor model.

3.1 Comparison

In Section 2 we reviewed the essential concepts in RESTful interfaces and in the actor model of computation. Table 1 shows the suggested correspondence. In the following, we investigate each pair individually.

Actor/Resource. Both actors and resources are meant to denote an entity that is self-contained and isolated. Moreover, both actors and resources are the basic components fundamental to the respective approach. In addition, the external interface of the constructed system is defined in terms of actors or resources. In RESTful services the resources available define the vocabulary for the web service, and in an actor system the *external actors* – those initially visible outside the system – determine the entities to which messages can be sent.

Mailbox/URL. In the actor model, the actors are identified by their mailbox addresses. Similarly, a RESTful design uses URLs to denote the resources the service exposes. In both scenarios the mailbox address or the URL is the only way for an external client to interact with the system. Moreover, both approaches allow entities that forward the received messages to other addresses. This corresponds to the use of aliases (e.g. `/blog/latest/`).

Acquaintance/Hypertext Link. In the actor model a node can only communicate with another node if it has its address. When the system is initialized, some set of *external* addresses is typically declared. In this way the actors whose addresses are revealed comprise the external interface of the system. When outside actors communicate with the external interface, they may receive addresses of actors that are not part of the external interface. Thus the topology of actor connections is not static but grows when the system is used.

Table 1. Counterparts in Actors and REST

Actor model	REST
Actor	Resource
Mailbox name	URL
Acquaintance	Hypertext link
Message	HTTP request
Behavior	Resource state change
Customer	Client

In the context of RESTful services, this behavior of actors giving out addresses of other actors corresponds to returning URLs in the resource's representation. This is especially important as now the evolving graph of actor's acquaintances makes the concept of hypermedia as the engine of application state explicit.

Messages/HTTP Request and Response. The actor model is based on asynchronous message passing with guaranteed delivery. REST, on the other hand, is built on top of HTTP and thus uses the traditional request/response paradigm. This might seem like a mismatch but, if necessary, the synchronous behavior of HTTP can be emulated by actors.

With respect to messages and their processing, the actor model is a lot more general. In the actor model of computation the messages can in principle be of any kind. HTTP, on the other hand, specifies pre-defined set of methods and return values. From the correspondence between the actor model and REST, this means that while using actors for analyzing RESTful services, we must limit ourselves to those of HTTP.

Behavior/Resource State Change. In the actor model, the functionality of an actor is defined by its behavior. This behavior changes over time as the actor processes messages and new replacement behaviors are decided. In REST, a resource has state. Also this state changes over time as requests are processed. For example, if a resource receives a `PUT` message it has to update its internal state to reflect the new representation of the data it received.

Customer/Client. The term customer refers to the sender of the message in the actor model. This way it becomes an acquaintance of the actor that receives the message. In REST the customer refers to the client of the request, which is most often the browser. In the actor model a "return value" of a behavior simply means sending a message to the customer. This is analogous to RESTful approach, where a response message always follows a request message. The only difference is that in actor model the response message is not mandatory.

3.2 Example

We now take a simple blogging service as an example and apply the ideas presented above. To keep the example simple, we omit issues such as user accounts, authentication, blog comments, and so forth. Thus the system under investigation consists solely of blog posts and their relationships.

We can consider the blogging service in two distinct ways. First, we can consider it to be a RESTful web service. On the other hand, we can start to analyze the service as a network of actors that process the messages of HTTP. To properly illustrate the relationship, we show both sides.

In REST, and with actors, we must first decide on the resources and then move on to the behavior. Since our service is simple, we only have a few resources. The root resource is the container for all blog entries. We give the name `/blog` for this resource. From the actor point of view, this means that the actor responsible

Table 2. Supported messages and their corresponding behaviors

	`/blog`	`/blog/2010-01-10`
`POST`	Create actor `/blog/2010-01-10` with default behavior and send 201 to customer	No effect, send `405` to customer
`GET`	Send `200` with addresses of the available blog entries (actors) to the customer	Send `200` with the contents of the blog entry to customer
`PUT`	No effect, send `405` to customer	Change the behavior of the actor to reflect the updated content. Send `200` to customer.
`DELETE`	No effect, send `405` to customer	In future forward all messages to actor representing status `404`. Send `200` to customer

for all posts should have a mailing address `/blog`. When new entries are added, they become new resources having names (addresses) like `/blog/2010-01-10` and `/blog/2010-01-15`. In the actor model this means that new actors are created whose addresses correspond to the URLs.

Because RESTful interfaces always implement a subset of the uniform interface instead of defining methods on the interface, we specify the supported set of the HTTP methods. This is traditionally done using a table that specifies the resource, method and the intended effect. For our example, such table is shown in the Table 2. Note that this time we formulate the intended effect in the language of actors. Numerical return codes used in the table are from the HTTP specification.

4 Applying Actors to a RESTful Interface

In order to better examine the relationship between the actor model and REST we need a notation for defining actor based RESTful web services. Using the notation we are then able to apply it to some use cases to gain better understanding on how the relationship would work in practice.

4.1 The Notation

Notations for actor systems are usually full fledged programming languages such as PLASMA [10] and Act2 [17]. However, also simpler alternatives exist. The minimalistic example in Listing 1.1 shortly demonstrates the notation used by Agha in [3]. This is not a programming language but a notation used to illustrate actor behaviors. The example defines a piggy bank actor that can be used to deposit money but one is able withdraw money only by breaking it. When the actor receives a `deposit` message, it replaces its behavior with the new balance. Conversely, when the actor receives a `break` message, it sends the balance to customer and is replaced by a behavior that ignores all messages (`sink`).

```
piggy_bank with acquaintance balance
  if message is deposit
    become new piggy_bank with balance + amount
  if message is break ∧ balance ≠ 0
    send amount to customer
    become sink
```

Listing 1.1. Piggy Bank Actor

```
piggy_bank[balance] =
  POST[amount] →
    send 200 to customer
    become (new piggy_bank(balance + amount))
  GET →
    send 405 to customer
  PUT →
    send 405 to customer
  DELETE →
    send balance to customer
    become 404
```

Listing 1.2. RESTful Piggy Bank

Obviously, the example only covers a small subset of all the features in the actor model, but it does portray a notation that can be used to describe an actor and its behavior. In the listing the actor has the balance as its only acquaintance as indicated by the `acquaintance` keyword. The balance becomes actor's acquaintance when the actor's behavior is created. The replacement behavior is defined using the `become` keyword.

From Agha's notation we derive our own RESTful actor notation. This notation is presented in Listing 1.2, which demonstrates a similar piggy bank service as the previous example. The difference is that this time the actor can be thought of as a RESTful web service. Our notation resembles Agha's notation but also shows clear connection to REST via its HTTP keywords. In the listing, `piggy_bank` defines the behavior of the actor. It has one acquaintance as denoted by the variable `balance` within brackets. The actor implements handlers for all four messages but it only supports the `POST` and `DELETE` messages. When a `POST` message is received, the actor sends `200 OK` to customer, whereas when a `DELETE` message is received, the balance is sent to customer after which the resource becomes unavailable. Other messages are responded with `405 Method Not Allowed`. The keyword `become` is omitted in cases where the replacement behavior is the same as the one being executed.

4.2 Example

By now we have established the analogy between the actor model and REST. We have also defined the notation for applying the actor model to a RESTful interface. Hence, we have the elements to take the blog example presented in Section 3.2 and demonstrate how to depict it using our notation.

```
blog[list] =
  GET →
    send 200[self/latest] to customer
  PUT →
    send 405 to customer
  POST[c] →
    new blog_entry(call POST[c] to list) @ self/current-date
    send 201[self/current-date] to customer
  DELETE →
    send 405 to customer

blog_entry[item] =
  GET →
    send 200[call GET to item] to customer
  PUT[c] →
    send PUT[c] to item
    send 200 to customer
  POST →
    send 405 to customer
  DELETE →
    send DELETE to item
    send 200 to customer
    become 404

# create the blog and bind it to an address
new blog(new list) @ /blog
```

Listing 1.3. Blog example

The Listing 1.3 defines two actor behaviors: `blog` and `blog_entry`. The `blog` defines the top level actor for all `blog_entry` actors. For the sake of brevity we have left out code listings for `list` and `list_item` actor behaviors, which are acquaintances of `blog` and `blog_entry` respectively. The list actor is responsible for storing the blog content. There are a couple of new notations used in Listing 1.3:

- `self` refers to the address of the actor instance itself.
- Symbol `@` makes actor external by binding it to the given address.
- `call` is similar to `send` but synchronous. That is, the actor blocks the execution until it has received the response from the actor it `call`s. The *call expression* of the actor model is defined in [3].

Next, we examine the most interesting parts of the Listing 1.3.

`blog.GET`: The `self/latest` is the address of the latest blog entry in the blog. Interestingly, messages sent to this address end up in the mailbox of different actors over time. Note that this behavior differs from the one presented in Table 2.

`blog.POST`: The `call POST[c] to list` sends `POST` message to the list and synchronously waits for the list item that is returned by the list. The list

item is then given as acquaintance to the new `blog_entry` actor. The new blog entry is bound to blog's own address appended with current date. The `current-date` is expected to be a primitive returning current date.

`blog_entry.GET`: The actor retrieves the contents of the blog entry synchronously from the `list_item` actor. Next, it sends `200` with the content to the customer.

`blog_entry.DELETE`: The actor deletes the contents of the blog entry by sending `DELETE` to its list item. Furthermore, it creates a replacement behavior `404` for itself.

The last line in the listing creates the blog and binds it to address `/blog`. Also note that the created list actor is not explicitly bound to any address. This means that it is not an external actor but visible only for the `blog` actor.

5 Discussion

The previous sections have motivated the relationship between the actor model of computation and RESTful services. Moreover, we have investigated the relationship in more detail and provided an example. In this section we discuss the pros and cons of the suggested approach to understand RESTful services as a network of actors.

5.1 Resources and Communication

The presented correspondence between REST and actor model of computation puts emphasis on resources and actors, message passing in the form of a subset of the uniform interface, and revealing the hypertext as an engine of application state via returned actor addresses. However, it is important to note what is abstracted away: selection of representation, cookies, and HTTP protocol headers.

At this abstraction level the emphasis is on the resources and actors themselves. Therefore it helps the designer to see the system being built as a network of entities with their internal state and behavior. As it also abstracts away implementation details such as databases, we must model the information storing using actors. This makes the concurrent nature of web services more visible as opposed to hiding it by delegating the concurrency problems to the database.

The use of actors also provides us with a built-in mechanism for inter-resource communication. So far, this has been a property of the framework actually used to implement a service. Also in these cases often the database has been used as an arbitrer to store the shared information. The actor view of RESTful services, on the other hand, does not make a distinction between local and external actors. Hence, external web services can be thought of as being actors too and they can be accessed using the same message passing paradigm as local actors.

5.2 Naming

The naming of resources poses a problem for our approach. In the original actor model the names of individual actors are opaque. That is, there is no external

representation that can be resolved by some mechanism to an actor. Naturally, when the addresses of actors are represented by URLs, there is an implicit assumption that a suitable URL corresponds to an actor. Indeed, the presented relationship builds on this assumption.

The opaqueness of actor naming also means that in our notation we have to provide means by which an actor is bound to a URL. The universal actor model [19] has investigated the naming of actors, but their model is not directly suitable for us, as it relies on an own naming scheme. The problem is made more difficult by the fact that when names are identified by URLs, there is nothing that stops the client from guessing arbitrary URLs and seeing whether they resolve to actual actors.

The most difficult problem related to naming occurs in conjunction with the `PUT` method. From a RESTful point of view, when a new resource is created using `PUT`, the distinction between a `PUT` and a `POST` is that a `POST` is targeted towards an existing resource whereas with `PUT` the client is in charge of naming the new resource. When considering the service as a network of actors, the `POST` case is easy: there exists a corresponding actor that in due course creates more actors if necessary. With `PUT`, however, there is necessarily no existing actor with the given URL and the actor model of computation does not allow an infinite number of actors.

5.3 Notation

In order to discuss RESTful systems as a network of actors, we have presented an informal notation. Although in the actor model there are no restrictions on the messages actors may send and receive, we must limit our notation to include only the ones supported by HTTP – the HTTP methods and status codes. In addition, the actor model imposes no predefined semantics for actor behavior. In contrast, RESTful actors must adhere to semantics of HTTP methods when responding to messages. This means, for example, that an actor is not allowed to create new actors when receiving a `GET` message.

As mentioned, we abstract away issues such as the content type of the representation. In addition, the notation is meant to be suggestive and it is not specified formally. Nevertheless, the notation puts emphasis on the inherent concurrency available when implementing RESTful services. The notation shows when it is possible to decide on the new behavior and to create a new actor machine respectively. Most important benefit of the notation is that it provides a tool for the developer to discuss a service especially from the point of view of REST, without implementation details.

6 Related Work

Actors have previously been used to analyze distribution in web applications [6]. Closer to our approach, however, is the actor based research done on middleware systems. Especially the research on Worldwide Computing Middleware [2] and the related research on universal actor model [18] is related to our work.

Although the Worldwide Computing Middleware (WCM) has identified many of the same connections between HTTP and actors (see the Table 1.1 of [2] and our Table 1) as we have, there are crucial differences in the overall approach. The WCM has a more generic approach and considers issues such as mobility that are outside the scope of this paper. Furthermore, the system they envisage would work outside the Web proper, although utilizing many of the familiar web concepts. Our approach, on the other hand, focuses strictly on RESTful services.

Research related to understanding RESTful interfaces has also been done in the context of modeling [12] [13]. However, the emphasis is placed on the process of developing a model that has the required RESTful properties. Our approach, on the other hand, investigates REST in a context of a computation model. This model of message passing has been previously suggested for web services [5], but not specifically, as far as we know, in the context of REST.

7 Conclusions

In recent years, REST has gained popularity as an architectural style of choice for big and complex web services. Unfortunately many of these services fall short of truly embracing the REST principles. To alleviate this problem and to gain better understanding of RESTful services in general, we need solid foundation as to how RESTful web services should be designed and modeled.

In this paper we have established the relationship between the actor model of computation and REST. We have presented the analogy in detail and identified the mismatching features between the two approaches. In order to apply the actor model to a RESTful web service, we have defined a notation for it. Using this notation we have designed a simple blogging service to illustrate the usefulness of the approach.

Building on the work reported in this paper, our next objective is to formalize the notation and build an environment where we are able to visualize the network of actors created by their behaviors. We hope that this visualization will help us to better understand complex RESTful web services.

References

1. Agha, G., Thati, P., Ziaei, R.: Actors: A Model For Reasong About Open Distributed Systems. Formal Methods for Distributed Processing - An Object Oriented Approach, ch. 8. Cambridge University Press, Cambridge (2001)
2. Agha, G., Varela, C.A.: Worldwide computing middleware. In: Singh, M. (ed.) Practical Handbook on Internet Computing. CRC Press, Boca Raton (2004) (invited book chapter)
3. Agha, G.A.: Actors: A model of concurrent computation in distributed systems. Technical Report 844, MIT Artifical Intelligence Laboratory (June 1985)
4. Armstrong, J.: Programming Erlang: Software for a Concurrent World. The Pragmatic Bookshelf (June 2007)
5. Böhm, A., Kanne, C.-C.: Processes Are Data: A Programming Model for Distributed Applications. In: Vossen, G., Long, D.D.E., Yu, J.X. (eds.) WISE 2009. LNCS, vol. 5802, pp. 53–56. Springer, Heidelberg (2009)

6. Chang, P.H., Agha, G.: Supporting reconfigurable object distribution for customized web applications. In: The 22nd Annual ACM Symposium on Applied Computing, SAC (2007)
7. Erenkrantz, J.R., Gorlick, M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of web architectures. In: ESEC-FSE '07: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering, pp. 255–264. ACM, New York (2007)
8. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (June 1999), updated by RFC 2817 `http://www.ietf.org/rfc/rfc2616.txt`
9. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
10. Hewitt, C.: Viewing control structures as patterns of passing messages. A.I. Memo 410, MIT Artifical Intelligence Laboratory (December 1976)
11. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, pp. 235–245. Morgan Kaufmann Publishers Inc, San Francisco (1973)
12. Laitkorpi, M., Koskinen, J., Systa, T.: A uml-based approach for abstracting application interfaces to rest-like services. In: WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering, pp. 134–146. IEEE Computer Society Press, Washington (2006)
13. Laitkorpi, M., Selonen, P., Systa, T.: Towards a model-driven process for designing restful web services. In: ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services, pp. 173–180. IEEE Computer Society Press, Washington (2009)
14. Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. "big" web services: making the right architectural decision. In: WWW '08: Proceeding of the 17th International Conference on World Wide Web, pp. 805–814. ACM, New York (2008)
15. Razavi, A., Marinos, A., Moschoyiannis, S., Krause, P.: RESTful Transactions Supported by the Isolation Theorems. In: Gaedke, M., Grissnikalus, M., Diaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 394–409. Springer, Heidelberg (2009)
16. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly, Sebastopol (2007)
17. Theriault, D.G.: Issues in the design and implementation of act2. Technical Report 728, MIT Artifical Intelligence Laboratory (June 1983)
18. Varela, C.: Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination. Ph.D. thesis, U. of Illinois at Urbana-Champaign (2001)
19. Varela, C.A., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. In: ACM SIG-PLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings, vol. 36(12), pp. 20–34 (December 2001)
20. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR, Upper Saddle River (2005)