Computational REST Meets Erlang

Alessandro Sivieri, Gianpaolo Cugola, and Carlo Ghezzi

Politecnico di Milano
DeepSE Group, Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 Milano, Italy
{sivieri,cugola,ghezzi}@elet.polimi.it

Abstract. Today's applications are developed in a world where the execution context changes continuously. They have to adapt to these changes at run-time if they want to offer their services without interruption. This is particularly critical for distributed Web applications, whose components run on different machines, often managed by different organizations. Designing these programs in an easy and effective way requires choosing the right architectural style and the right run-time platform. The former has to guarantee isolation among components, supporting scalability, reliability, and dynamic changes. The latter has to offer mechanisms to update the applications' code at run-time.

This work builds upon previous research about architectures and run-time platforms. Its contribution is to put together a very promising architectural style – Computational REST – with a language (and run-time environment) designed with dynamic, distributed applications in mind – Erlang. We show how they fit together by developing a new framework, which eases development of highly distributed Web applications capable of operating in dynamic environments. We also provide an initial experimental assessment of the proposed approach.

Keywords: Computational REST, Erlang, OTP, architectural styles, programming languages, mobile code, Internet.

1 Introduction

The technological evolution in networking has changed the way applications are designed and developed: instead of having monolithic programs created for desktop computers running in isolation, more and more often we have large-scale, distributed Web applications, whose components run on many different devices, from personal computers to smartphones, from mainframes to low-power sensors. In the most challenging scenarios, these applications put together components built and administered by different organizations, by invoking the services offered by such components, managing the data that flow among them and using a browser as the front-end.

To further complicate things, these Web applications are usually expected to run for long time without interruption and failures: a challenging goal if we consider that the devices and components they are built upon may change

J. Bishop and A. Vallecillo (Eds.): TOOLS 2011, LNCS 6705, pp. 244–259, 2011.

over time in a way that is often hard to forecast. Software Engineering is asked to address this issue by developing ad-hoc programming frameworks to ease the implementation of largely distributed Web applications capable of handling changes (and failures) in the external services they invoke and in the devices they access and run on, in a smooth and effective way.

Such programming frameworks should integrate an architectural style that guarantees isolation among components, supporting scalability, reliability, and dynamic changes, with a programming language (and run-time support environment) that offers mechanisms for dynamic update of functionalities.

Current research has proposed Computational REST (CREST) [14], as an effective architectural style to build dynamic, Internet-wide distributed applications. CREST extends the REpresentational State Transfer (REST) style [16], changing the focus from *data* to *computations*, while maintaining the REST principles, which guarantee Internet-wide scalability. In a CREST application, each component (called *peer*) is able to exchange computations, in the form of *continuations* or *closures*, with other components, to dynamically install new services on remote components and demand their execution to others. This idea of managing computations as first-class elements comes from the research on mobile code [17], and has proved to be an effective mechanism to easily support dynamic changes for long-running applications.

While CREST is just an architectural style, their authors proposed a prototype programming framework that embeds the CREST principles in Scheme [12], a well-known functional programming language. This choice was motivated by the Scheme capabilities in dealing with continuations, which allow Scheme processes to be easily suspended to be resumed later. On the other hand, Scheme does not offer any native support to building distributed applications, a critical aspect for a framework that has distributed applications as its main target.

Starting from this consideration we decided to see if other languages could better fit the CREST principles. In this we were also motivated by the fact that the original Scheme-based prototype was never made officially public, at least not in a form that allow it to be used in practice for experiments.

In particular, we chose Erlang [5], a functional language that was designed upfront to build long-running distributed applications. Indeed, Erlang and its OTP [5] library natively support distributed programming, offering advanced and easy-to-use mechanisms to remotely spawning components, letting them communicate, and automatically managing failures. In addition, it offers mechanisms to dynamically change the components' code at run-time. These features are embedded in a functional core that supports closures, a fundamental aspect to satisfy the CREST requirements.

The rest of the paper describes the result of our experience. In particular, Section 2 introduces Computational REST and Erlang. Section 3 describes the new Erlang-based CREST framework and the facilities it provides to developers, while Section 4 compares it against the original CREST framework and a pure REST (i.e., Web-based) implementation of the same application, in terms of performance, functionalities offered, and cost to implement them. Finally

Section 5 discusses related work and Section 6 draws some conclusions and suggests possible future work in the area.

2 Background

In this section we briefly introduce the main topic areas upon which our work is based, i.e., the REST and CREST architectural styles and the Erlang language.

2.1 The REST and CREST Styles

Defined by R.T. Fielding (one of the main authors of the HTTP protocol), the REpresentational State Transfer (REST) style provides an *a posteriori* model of the Web, the way Web application operates, and the technical reasons behind the Web success.

Fielding's Ph.D. thesis [16] defines the set of *constraints* that every REST application should satisfy: the structure of the application has to be client-server, communication has to be stateless, caching has to be possible, the interface of servers has to be standard and generic, layering is encouraged, and each single layer has to be independent from the others. An optional constraint suggests using code-on-demand [17] approaches to dynamically extend the client's capabilities.

These constrains are coupled with a set of *foundation principles*:

- the key abstraction of information is a resource, named by a uniform resource identification scheme (e.g., URLs);
- the representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes;
- all interactions are context-free;
- only a few primitive operations are available;
- idempotent operations and representation metadata are encouraged in support of caching;
- the presence of intermediaries is promoted.

While these principles allowed REST to be scalable and supported the current Web dimensions, at the same time not all the Web applications followed these design guidelines; for example, they might require stateful communications or they might create problems to caching devices components.

The main limitation of REST is the generic interface constraint: it improves independence of applications on specific services, because all the components are able to handle any data, but at the same time it hampers the efficiency of communication, since all data must be coded in a standard way to pass through standard, application independent interfaces; something not easy to do especially when there is more than pure "content" to be sent between peers.

The CREST authors identified this and other REST weaknesses in [13] and decided to address them by moving the focus of the communication from *data* to *computations*. If the former is the only subject of an interaction, then a client

receiving a message through a generic interface could not be able to interpret it correctly. The REST optional constraint of code-on-demand is too weak to solve the issue, since the same client could not be able to use that code.

The result of this paradigm shift was the Computational REST (CREST) [14] style, which let *peers* exchange *computations* as their primary message, usually implementing them through continuations. These are instances of computations suspended at a certain point and encapsulated in a single entity to be resumed later. They are offered as a basic construct by some languages, usually functional ones like Scheme, which also allow continuations to be serialized and transmitted along a network connection to allow the computation to be resumed on a different node.

Whenever a language does not offer the continuation mechanism, a *closure* can be used instead: it is a function with free variables declared within its scope, and since the extent of these variables is at least as long as the lifetime of the closure, they can be used for saving a state between different calls of the function. Later, in Section 3 we will explain why using this less powerful mechanism instead of continuations does not influence the expressiveness of our framework.

Also notice that in the definitions above we used the term "peer" instead of "client" or "server". This is not by accident, since CREST does not distinguish between clients and servers but rather between *weak peers* that support a minimal subset of the CREST operations and usually operate as initiators of the interaction, and *strong peers* that support the whole set of CREST operations and characteristics and may fully interact with other peers, be they strong or weak.

CREST draws on the REST principles to define a new set of architectural guidelines:

- a resource is a locus of computations, named by a URL;
- the representation of a computation is an expression plus metadata to describe the expression;
- all computations are context-free;
- the presence of intermediaries is promoted;
- only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.

As for the last point, CREST defines two primitive operations: the *spawn* operation requires the creation of a process executing the computation; this process is associated to a unique URL and when this URL is invoked the computation itself is resumed and the results it produces are returned to the caller; thus, new services can be installed in a (strong) peer and then accessed by any client. The *remote* operation installs a computation and resumes it immediately, returning any result to the caller and destroying it when it ends, so that it cannot be accessed again.

In [13,19] the authors further detail the CREST principles:

- any computation has to be included into HTTP operations, so that the new paradigm could be made compatible with the current Internet infrastructure.

To keep up with such compatibility, the authors also distinguish between machine URLs and human-readable URLs, where the former may contain the computation itself, while the latter can be used by users;
- computations may produce different results, based on any received parameter, server load or any other factor that changes during time; they can also maintain a state between calls, for example for accumulating intermediate results;
- computations have to support independency between different calls, and avoid data corruption between parallel invocations using synchronization mechanisms offered by the languages of choice;
- computations can be composed, creating mashups: a computation may refer to other computations on the same peer or on different peers, and an execution snapshot should include the whole state of the computation;
- intermediaries must be transparent to the users;
- peers should be able to distribute computations, to support scaling and lowering latency, also checking temporal intervals between executions of the same computation and specifying some sort of expiration date when necessary.

Finally, in [19] a new feature has been introduced: spawned processes should act as so-called *subpeers*, with their own *spawn* and *remote* capabilities, inheriting security policies by their ancestors in the process tree, where the root node is the peer itself. This way a hierarchy of processes is created in a CREST peer, where each node is limited by its ancestors and limits its successors.

Security concerns. An important issue with architectural styles for distributed applications is security. Besides traditional security concerns, the CREST adoption of mobile code technologies opens new problems; namely how to secure the peer against the code it receives and how to secure the code against the peer in which it is executed [29,30]. The CREST definition recognizes the issue but provides few details on how to address it. In practice, the current CREST framework, implemented using a Scheme interpreter running into a Java Virtual Machine, leverages the sandbox mechanism of Java, using an ad-hoc Security Manager that limits the resources accessible to the incoming computations. Moreover, the authors suggest that the bytecode received by a peer should be inspected and checked for instruction sets executing commands that are not allowed by the (sub)peer security policy, while self-certifying URLs [23] could be used for mutual authentication between peers.

2.2 Erlang

Erlang [3,4,5,6] is a programming language originally defined to implement parallel, distributed applications meant to run continuously for long periods[1]. It provides a set of features that make it a perfect choice for a framework to build CREST-compliant applications.

[1] The definition of Erlang has been primarily motivated by the requirements of telecommunications applications within Ericsson.

In particular, its functional language core natively supports *closures*, which – while not offering the full expressive power of continuations – are a step in the right direction to implement the CREST idea of exchanging computations among peers. Moreover, Erlang combines dynamic typing and the use of pattern-matching as the main mechanism to access data and guide the computation, supporting a form of declarative programming that allows programmers to focus on *what* a computation is supposed to do instead of *how* to achieve it. This results in extremely compact code that is easy to develop and maintain. We found these features fundamental to develop a programming framework that has to be open to extensions by application programmers who wish to build their own, CREST-compliant software.

In addition, Erlang enriches its functional core with ad-hoc language constructs to build parallel and distributed applications. In particular, Erlang uses an actor-like concurrency model [22], which allows for easily and naturally organizing every Erlang computation as a (large) set of light processes, automatically mapped by the Erlang runtime into system threads and hardware cores. Since such processes cannot share memory and have to rely on message passing (which is embedded into the language) to communicate, this approach also naturally supports developing distributed applications, another fundamental feature to ease the implementation of our CREST framework.

A further peculiarity of Erlang is the fact that its runtime support system allows application code to be hot-swapped. This mechanism was introduced to support long running applications, like those implemented into telephonic switches, and can be used as a way to change the code of an application at runtime without interrupting it. In particular, if a module function is executed by calling its qualified name, then the runtime guarantees the execution of the last version available of that function; that is, if the module bytecode is updated while the application is running, then each new function invocation will use the last version of the code, while any running instance will continue its execution with the previous one. Notice however that only two versions of a module may live together at the same time: if a third one is added, then the second one becomes the "old" one and the first one is dropped, and each computation using it is automatically killed.

Finally, Erlang provides an extensive, standard library, called the Open Telecom Platform (OTP), which offers predefined modules for process linking and monitoring. By using OTP, supervision trees of processes can be easily constructed so that each supervisor is able to monitor if a process crashes and restart it or propagate the error. OTP also offers several modules, called *behaviors*, which implement the non-functional parts of a generic server so that a developer can focus only on the functional ones. Altogether, these functionalities greatly simplify the development of fault-tolerant applications, and we leveraged them to reduce the effort needed in implementing our CREST framework.

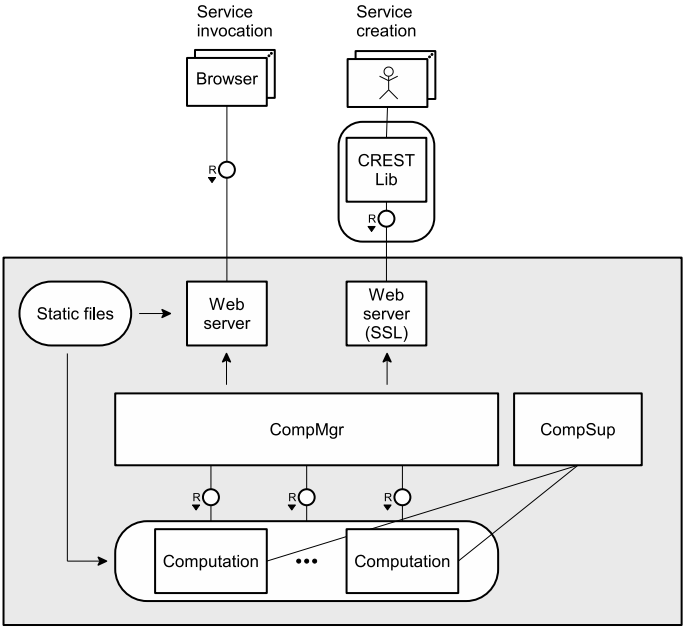


Fig. 1. Server structure

3 CREST-Erlang

In this section we illustrate how the CREST style can be implemented in Erlang. The resulting framework is called CREST-Erlang, as opposed to CREST-Scheme, which denotes the original framework presented in [13]. Figure 1 shows the structure of a CREST peer written in Erlang. At the bottom are the computations running into the peer, which have been installed there by invoking the *spawn* or *remote* CREST primitives. They are managed by an ad-hoc component, the `CompMgr`, which installs new computations, keeps a list of those running inside the peer, and dispatches incoming invocations.

As we mentioned, one of the main reasons to choose Erlang was the support offered by the language to let (distributed) processes communicate. On the other hand, to be CREST compliant, the communication among peers has to use the HTTP protocol. Accordingly, our peer embeds a Web server, which waits for incoming HTTP requests, unmarshals them, and uses the standard Erlang communication facilities to dispatch them to the `CompMgr`. More precisely, Figure 1 shows two Web servers, one answers HTTPS requests and is meant to handle *spawn* and *remote* operations, which we choose to securely transfer on top of SSL (more on this later). The other serves standard invocations and static pages, a trivial but required functionality for a Web framework.

As for the adopted protocol we chose, it is worth mentioning here that we decided to send computations using the HTTP POST operation, while the original

CREST approach suggests embedding them into the URL of the *spawn* request. This choice seems more in line with the expected usage of HTTP. Indeed, the POST operation has been designed for those requests that are expected to alter the internal state of the receiving server, and this is the case for the installation of a new service. Moreover, the POST payload may include a large body of data, as it happens in the case of the state of a computation and the associated bytecode.

As shown in Figure 1, our framework also includes the `CRESTLib`, which provides a set of facilities to invoke local and remote services without having to bother with the underlying communication details. This is used by peer clients, but it can also be used to implement the services themselves, when they have to communicate with other peers.

Finally, to improve fault tolerance each peer is organized in a supervision tree, with a high level supervisor (not shown in figure) in charge of all the fundamental modules including the two Web servers and the `CompMgr`, and a low level one, the `CompSup`, to which all the spawned computations are attached. The former is able to monitor and restart each of its children, while the latter, at the current state, just logs any error or exception happening to computations, unlinking them from the `CompMgr` when this happens.

Listing 1.1. Service template

```
1  my_service(State) ->
2    receive
3      {Pid, [{"par1", P1}, {"par2", P2}, ...]} ->
4        %% Do your job accessing par1, ... parN
5        %% eventually create a new state NewState
6
7        %% If necessary, spawn myself on peer Hostname
8        invoke_spawn(Hostname, ?MODULE,
9                     fun() -> my_service(NewState) end),
10       %% Finish with a tail recursion (or just end this
11       %% computation)
12       my_service(NewState)
13   end.
```

Listing 1.1 shows the template of an Erlang *service* to be spawned or remotely executed on a peer. It receives from the `CompMgr` the invocation parameters originally coming from the client, uses them to perform its computation, and finishes by invoking itself with the new state calculated during execution, using the typical approach of functional programming based on tail recursion. Lines 8-9 show how the service may spawn a copy of itself (i.e., a copy of the computation) on a different node, if necessary.

Notice that what is transferred to the other peer through the `invoke_spawn` primitive is the *closure* of the running service, not the *continuation*, as required

by CREST. Indeed, as we mentioned in the previous section, this is the only primitive offered by Erlang. On the other hand, the need to transfer computation while it is executing statements in the middle of the service's code is very uncommon. The typical service pattern is the one shown by our template, which transfers the computation just before recursing. If this is the case, transferring the closure obtains the same result as transferring the continuation of the computation.

Technologies involved and details about security. For the Web server part, we analyzed several different platforms developed in the last few years for handling HTTP communications in Erlang. Each has its pros and cons, and in the end we chose *MochiWeb* [1], because of its support to JSON (which we used to effectively serialize parameters and return values passed among peers and clients) and RESTful services, and for its performance.

The MochiWeb library and the OTP modules together provide the main skeleton of our peer: the supervising system, the logging system (not shown in Figure 1), and the two Web servers. This allowed us to focus on developing the functional parts of the framework.

As for security, Erlang does not offer many facilities. Indeed, it was born as a language for handling telephony devices, a domain in which security is usually guaranteed by directly controlling the network itself. Now that Erlang is being used outside its target domain, this weakness has been identified and the first security facilities are being added to the language. On the other hand, we are far from having ad-hoc facilities to manage security in general and the security of mobile code in particular. To address this issue we decided to adopt a strategy based on mutual authentication among peers. This way we bypass the specific problem of protecting the incoming computation from the peer and the peer from the computation, building a trusted network on top of which computations may roam freely. This is clearly a sub-optimal solution, which we plan to overcome in future versions of our prototype.

4 An Assessment of CREST-Erlang vs. CREST-Scheme

In this section we discuss how our CREST framework, based on Erlang, can be compared with the original CREST-Scheme solution. To perform the assessment, we chose to focus on three main dimensions: whether the same functionalities are offered by both, the cost in implementing them, and how they perform.

Functionalities. The original CREST-Scheme framework includes a case study to show the potential of the new approach, namely a shared RSS reader. It includes an AJAX Web site as a front-end, with several widgets to show the news (coming from a given RSS feed URL) using different visualization techniques. Each widget type interacts with a different service (i.e., computation) on a single CREST peer, while different instances of the same widget type (running on different clients) share the same service. This way every client sees the same information about the feeds. A user may duplicate the whole application instance, so that its changes will be separated from the original one.

The drawback of this case study is that every CREST computation resides on the same peer and when new computations are spawned (i.e., when a client duplicates the application's session) they are spawned into the same peer. In other words there is no transmission of computations among peers.

Accordingly, we implemented an additional case study to evaluate our framework: a distributed text mining application. A network of computers, each running a CREST-Erlang peer, share a set of documents to be analyzed. A front-end Web application allows the user to choose the text mining function and the set of peers to use. The former is sent as a spawned computation on the involved peers, which perform their part of the job and return the results back.

Differently from the original one, this case study leverages all the CREST mechanisms: *spawn* and *remote* operations, statefull and stateless computations, and service composition. This allowed us to asses the correctness and ease of use of the new framework.

The only point not covered by our CREST-Erlang framework is the concept of *subpeer*, which has been described by the CREST authors in a subsequent article [19], so it was not included in the current prototype.

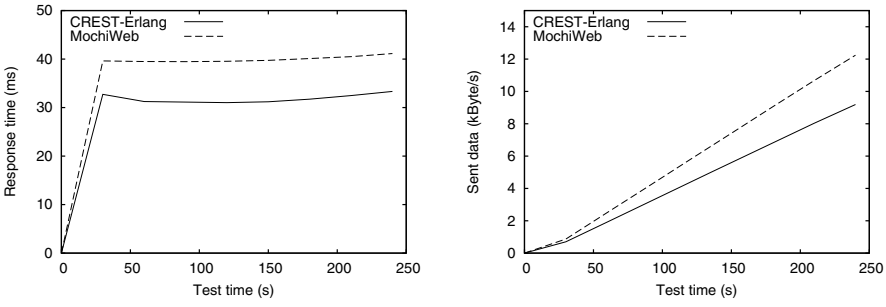
Table 1. Line code comparison

Framework	Framework source code	Demo source code
CREST-Scheme	5938	817
CREST-Erlang	2957	768

Implementation effort. To compare the effort in implementing the two framework, and so to indirectly compare the choice of the two languages used, i.e., Scheme vs. Erlang, we counted the lines of code of the main library and of the implemented case studies, not counting the external dependencies. The results are illustrated in Table 1 and show that our code is about a half of the original one. This fact confirms our initial idea that Erlang more easily and naturally supports the CREST mechanisms.

Performance measurement. To compare the two frameworks in terms of performance, we re-built part of the implementation of the original case study, in particular we used the same Web client application (with its graphical widgets) and recreated some of the corresponding CREST services. We also implemented this case study as a standard Web application using MochiWeb alone, to use as a reference. This was possible since the original case study, unless the client duplicates its session, does not exploit any advanced CREST functionality; all computations are installed during system startup, and they are only invoked at the demo.

To actually measure the performance of these three applications, we used a dual core laptop with 4GB of RAM as a server, and we launched several simulated users from a different computer, a 6 core desktop with 8GB of RAM. Notice that



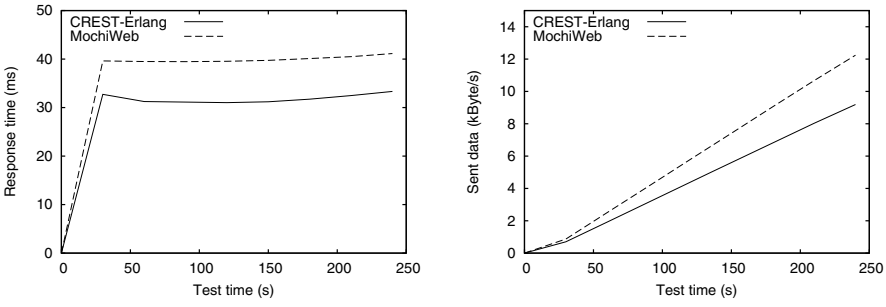
(a) Response time (b) Throughput

Fig. 2. CREST-Scheme demo

we choose the machine running the clients to be more powerful than the server to be sure the values we measured were not influenced by some limitation on the client side. The two machines are connected by a 100Mbit LAN. The whole test is run by using a client application, written in Erlang, which measures the average response time for each request and the throughput in term of KBytes per second sent to the clients. We used a navigation sample recorded during a browser session through the demo site to simulate the behavior of a standard user. Through our script we simulated the arrival of one of such users every second, each repeating the same session with a delay of one second at the end, for 4 minutes in total.

Figure 2 shows the results we measured in terms of response time and throughput. The CREST-Scheme framework has the worst performances, serving a very low number of pages per second with a response time peaking at more than 30 seconds; Mochiweb performs better than CREST-Erlang in terms of response time, because of the overhead introduced internally by the latter, and it is also able to answer more requests per second in the last minute of the test, because its usage of the server resources is lower than the CREST-Erlang one, especially in terms of CPU usage.

To test the overhead introduced by using the *spawn* and *remote* CREST operations, we compared our prototype against MochiWeb in running a Web application based on a simple CREST service. Each client starts by asking a front-end peer to spawn a new instance of this simple service on a different peer, located on the same machine, and from then on it invokes this new service repeatedly, with one second delay among each invocation; the MochiWeb version has the same service pre-installed, which the client invokes repeatedly as before. As in the previous case, we start one client every second for the 4 minutes of the test. Figure 3 illustrates the results we gathered in terms of response time and throughput. We notice that MochiWeb is able to answer more requests per second, and this explains the higher throughput, while the response time is similar and it remains almost constant while the number of clients increases.



(a) Response time (b) Throughput

Fig. 3. Test application

5 Related Work

The work we presented here is related with current research on evolvable and dynamically adaptable software architectures and on programming languages supporting dynamic adaptation. Seminal work on the identification of the critical architectural issues concerning run-time evolution is described in [25,28,26]. The CREST approach is largely motivated by this work. Several alternative architectural styles exist to support dynamically evolvable distributed applications. Hereafter we briefly review the most relevant ones and we contrast them with CREST.

Publish-subscribe (P/S) [15,9] is an event-based style where components are not directly connected, but communicate through a common middleware system, which takes any new event notification and dispatches it to any component subscribed for that specific event. This structure is highly dynamic since nodes may be added and removed while the system is running; communication is asynchronous and components can operate independently of each other.

Map-reduce (M/R) [10] is a style used to parallelize a computation over a large data set by distributing work over a collection of worker nodes. In the *map* phase each node receives from a master node some amount of data and elaborates it, returning key-value pairs to the master, while in the *reduce* phase the master node takes the answers to all the sub-problems and combines them to produce the output. Because worker nodes may be masters, a tree structure can be easily obtained, increasing scalability. As in the P/S case, M/R nodes are completely autonomous; they may join and leave dynamically as they do not share any data or state directly, and perform their computation in isolation w.r.t. the others.

Similarly to CREST, P/S and M/R architectural styles are oriented to dynamic adaptation, but differently from CREST they are not specifically oriented to supporting Web applications, probably the most important domain for distributed applications today and the one we target.

The two architectural styles that are today competing for becoming a standard in building Web applications are REST and the *Service-Oriented architecture* (SOA) [11]. We already discussed the differences between CREST and

REST in Section 2.1. SOA models a Web application as a composition of different autonomous services, independently developed and existing in different namespaces and execution contexts. Services may be dynamically discovered and compositions may bind to them dynamically. Usually these services operate over HTTP using Web Service protocols supporting standardized discovery and service invocation. Unfortunately, these protocols violate REST principles, as we already discussed in Section 2.1, and this can be a major problem, since REST principles are those that guaranteed the success of the Web.

CREST not only follows the REST principles, but also promises to support dynamic adaptation much better. Indeed, both REST and SOA focus on data as the primary element exchanged among components and this makes it hard to adapt the architecture of the application dynamically, since this usually requires to introduce new components/services. Vice-versa, CREST adopts the computations themselves as the elements exchanged among nodes (i.e., peers) and this makes it straightforward to change the architecture of the application at run time, when required.

Besides architectural styles, another research direction related with the work presented in this paper concerns programming languages. In particular, the identification of features or language constructs that may provide better support to the specific requirement of run-time adaptation. This sometimes leads to extensions of existing languages to support dynamic adaptation. For example, *context-oriented programming* extensions have been proposed and implemented for various languages [2], starting from initial work on LISP [8], up to the initial version of ContextErlang [18] developed by our research group. The features supported by Aspect-Oriented programming languages [24], and in particular Dynamic Aspect-Oriented languages [21], have also been proved to help in this context.

Functional programming languages, and in particular the notions of continuation and closure, have also been revamped in the context of Web programming. A short summary of work upon which CREST-Scheme is rooted can be found in [7], while examples of use of functional programming concepts in Web applications are provided in [20,27].

6 Conclusions

This article presented CREST-Erlang, a new implementation of a Web framework supporting the CREST architectural style. CREST is a promising style, which suggests to move from an Internet of *data* to an Internet of *computations* to cope with the dynamism of distributed applications developed nowadays.

As its name suggests, the new framework adopts Erlang as its reference language, while the original CREST framework adopted Scheme. This choice was motivated by the fact that Erlang provides advanced mechanisms to develop strongly concurrent, fault-tolerant, distributed applications in an easy and effective way. This intuition is confirmed by the experience reported in this paper. Erlang required less effort than Scheme to develop the framework and the resulting prototype performs better than the original one. Also, as we found easier to

develop the framework using Erlang than using Scheme, we argue that programmers using the framework to build CREST applications would benefit from a language that eases development of efficient algorithms, by natively supporting an effective form of concurrency (through the actor paradigm), which very well fits current multi-core hardware.

The main drawback we found was the limited support offered by the language and associated library to security, especially the peculiar form of security required when computations are expected to move among nodes. We provided an initial solution to the problem, but more has to be done.

As for our experience in using CREST, we found it an effective architectural style to build Web applications that could follow the somewhat natural evolution from an Internet of data to an Internet of computations.

On the other hand, a few remarks emerge from this experience. The first is about the protocol for CREST specific operations: is HTTP really the best protocol for transmitting computations? HTTP was developed for accessing documents. Although it is now often used as a general-purpose protocol, this was not its original purpose. Even in the case of Web Services, data had to be encoded in some document-like intermediate representation, such as XML, before being moved to clients, with a certain overhead. The same happens when computations, including state, function references, and code have to be transferred.

The second remark is about security. We already see in today's Internet the security issues induced by the code-on-demand features of Web 2.0 sites, with malevolent Javascript code used for stealing users' data. We can easily imagine what could happen if computations are allowed to move around on the backweb. Apart from citing the usual countermeasures, developed for mobile agent platforms and never assessed in realistic, large scale, open environments, the CREST definition does not provide any specific solution to this problem. We are convinced that this could severely limit the adoption of the CREST style until something new is developed.

As for our future plans, we want to continue developing our prototype, by introducing a caching mechanism that may further increase performance. We will also integrate the concept of *subpeer*, introduced in the latest CREST definition [19].

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom; and by the Italian Government under the projects FIRB INSYEME and PRIN D-ASAP.

References

1. Mochiweb, `http://github.com/mochi/mochiweb`
2. Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A comparison of context-oriented programming languages. In: COP 2009: International Workshop on Context-Oriented Programming, pp. 1–6. ACM Press, New York (2009)

3. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Ph.D. thesis, Royal Institute of Technology, Sweden (December 2003)
4. Armstrong, J.: A history of erlang. In: HOPL. pp. 1–26 (2007)
5. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (July 2007), http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN%/193435600X
6. Armstrong, J.: Erlang. Commun. ACM 53(9), 68–75 (2010)
7. Byrd, W.E.: Web programming with continuations. Tech. rep., Unpublished Tech. Report (2002), http://double.co.nz/pdf/continuations.pdf
8. Costanza, P.: Language constructs for context-oriented programming. In: Proceedings of the Dynamic Languages Symposium, pp. 1–10. ACM Press, New York (2005)
9. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. (to appear)
10. Dean, J., Ghemawat, S.: Mapreduce: a flexible data processing tool. Commun. ACM 53(1), 72–77 (2010)
11. DiNitto, E., Ghezzi, C., Metzger, A., Papazoglou, M.P., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Autom. Softw. Eng. 15(3-4), 313–341 (2008)
12. Dybvig, R.K.: MIT Press, 4th edn. MIT Press, Cambridge (2009)
13. Erenkrantz, J.R., Gorlick, M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of web architectures. In: ESEC-FSE 2007: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 255–264. ACM Press, New York (2007), http://dx.doi.org/10.1145/1287624.1287660
14. Erenkrantz, J.R.: Computational REST: a new model for decentralized, internet-scale applications. Ph.D. thesis, Long Beach, CA, USA (2009) Adviser-Taylor, Richard, N.
15. Eugster, P.T., Felber, P., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
16. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis (2000), http://portal.acm.org/citation.cfm?id=932295
17. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding code mobility. IEEE Transactions on Software Engineering 24, 342–361 (1998)
18. Ghezzi, C., Pradella, M., Salvaneschi, G.: Context oriented programming in highly concurrent systems. In: COP 2010: International Workshop on Context-Oriented Programming, co-located with ECOOP 2010, Maribor, Slovenia (2010) (to appear)
19. Gorlick, M., Erenkrantz, J., Taylor, R.: The infrastructure of a computational web. Tech. rep., University of California, Irvine (May 2010)
20. Graunke, P., Findler, R.B., Krishnamurthi, S., Felleisen, M.: Automatically restructuring programs for the web. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE 2001, p. 211. IEEE Computer Society, Washington, DC, USA (2001), http://portal.acm.org/citation.cfm?id=872023.872573
21. Greenwood, P., Blair, L.: L.: Using dynamic aspect-oriented programming to implement an autonomic system. Tech. rep., Proceedings of the, Dynamic Aspect Workshop (DAW04, RIACS (2003)

22. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, pp. 235–245. Morgan Kaufmann Publishers Inc, San Francisco (1973), http://portal.acm.org/citation.cfm?id=1624775.1624804
23. Kaminsky, M., Banks, E.: Sfs-http: Securing the web with self-certifying urls
24. Masuhara, H., Kiczales, G.: Modeling crosscutting in aspect-oriented mechanisms, pp. 2–28. Springer, Heidelberg (2003)
25. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE, pp. 177–186 (1998)
26. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: 30th International Conference on Software Engineering, pp. 899–910. ACM Press, New York (2008)
27. Queinnec, C.: The influence of browsers on evaluators or, continuations to program web servers. In: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 23–33. ACM, New York (2000), http://doi.acm.org/10.1145/351240.351243
28. Taylor, R.N., Medvidovic, N., Oreizy, P.: Architectural styles for runtime software adaptation. In: WICSA/ECSA, pp. 171–180 (2009)
29. Vigna, G. (ed.): Mobile Agents and Security. LNCS, vol. 1419. Springer, Heidelberg (1998)
30. Zachary, J.: Protecting mobile code in the world. IEEE Internet Computing 7(2), 78–82 (2003)