Node.Scala: Implicit Parallel Programming for High-Performance Web Services

Daniele Bonetta, Danilo Ansaloni, Achille Peternier, Cesare Pautasso, and Walter Binder

University of Lugano (USI), Switzerland
Faculty of Informatics
{name.surname}@usi.ch

Abstract. Event-driven programming frameworks such as Node.JS have recently emerged as a promising option for Web service development. Such frameworks feature a simple programming model with implicit parallelism and asynchronous I/O. The benefits of the event-based programming model in terms of concurrency management need to be balanced against its limitations in terms of scalability on multicore architectures and against the impossibility of sharing a common memory space between multiple Node.JS processes. In this paper we present Node.Scala, an event-based programming framework for the JVM which overcomes the limitations of current event-driven frameworks. Node.Scala introduces safe stateful programming for event-based services. The programming model of Node.Scala allows threads to safely share state in a standard event-based programming model. The runtime system of Node.Scala automatically parallelizes and synchronizes state access to guarantee correctness. Experiments show that services developed in Node.Scala yield linear scalability and high throughput when deployed on multicore machines.

1 Introduction

Services published on the Web need to guarantee high throughput and acceptable communication latency while facing very intense workloads. To handle high peaks of concurrent client connections, several engineering and research efforts have focused on Web server design [2]. Of the proposed solutions, event-driven servers [3,12] have proven to be very scalable, as they are able to handle concurrent requests with a simple and efficient runtime architecture [9,7]. Servers of this class are based on the ability offered by modern operating systems to communicate asynchronously (through mechanisms such as Linux's `epoll`), and on the possibility to treat such requests as streams of events. In event-driven servers each I/O-based task is considered an *event*. Successive events are enqueued for sequential processing (in an *event queue*), and processed in an infinite *event-loop*. The event-loop allows the server to process concurrent connections nondeterministically by automatically partitioning the time slots assigned to the processing of each request, thus augmenting the number of concurrent requests handled by the server through time-sharing. In this way, request processing is overlapped with I/O-bound operations, maximizing throughput and guaranteeing fairness between clients. Thanks to the event-loop model, servers can process thousands of concurrent requests using a very limited number of processes (usually, one process per core on multicore machines).

The performance of event-driven architectures has promoted programming models for Web service development that rely (explicitly or implicitly) on event-loops. Examples of such programming models include libraries (e.g., Python Twisted [6] or Java NIO [1]), and language-level integrations such as Node.JS [14]. Node.JS is a programming framework for the development of Web services using the JavaScript language and Google's V8 JavaScript Engine. In Node.JS the event-loop is hidden behind a convenient programming abstraction, which allows the developer to treat event-driven programming as a set of callback function invocations, taking advantage of the functional nature of the JavaScript language. Since the event-loop is run by a single thread, while all I/O-bound operations are carried out by the OS, the developer only writes the sequential code to be executed for each event within each callback, without worrying about concurrency issues.

Despite of the high performance of the V8 Engine, frameworks like Node.JS still present some limitations preventing them from exploiting modern multicore machines. For example, long running callbacks may block the entire service due to the single-threaded, sequential event loop architecture. We have overcome these limitations with the design of Node.Scala, a programming framework for the development of scalable Web services which takes full advantage of modern multicores. In more detail, our work makes the following contributions:

– We introduce Node.Scala, an event-loop-based framework targeting the Scala language and the JVM. Node.Scala features automatic parallelization of concurrent request processing, automatic synchronization of stateful request processing, and allows the developer to use both blocking and non-blocking programming styles. Node.Scala can be used to build HTTP-based services, including RESTful Web services [4].
– We describe the design of the Node.Scala runtime, which features multiple event-loops which have been safely parallelized.
– We illustrate the performance of Node.Scala with a set of benchmark results obtained with both stateless and stateful Web services.

The rest of this paper is structured as follows. In Section 2 we further discuss the motivations underlying Node.Scala and provide background information on event-loop frameworks. Section 3 presents the programming model of Node.Scala. Section 4 presents the parallel runtime system of Node.Scala. Section 5 presents an evaluation of the performance of Node.Scala-based Web services. Section 6 discusses related work, while Section 7 concludes.

2 Background and Motivation

Despite of being very scalable in terms of handling concurrent connections, event-driven frameworks like Node.JS are limited by their runtime system at least in two aspects, namely (1) the impossibility of sharing a common memory space among processes, and (2) the difficulty of building high throughput services using blocking method calls.

Concerning the first limitation, common event-based programming frameworks are not designed to express thread-level parallelism, thus the only way of exploiting multiple cores is by replicating the service process. This approach forces the developer to adopt parallelization strategies based on master-worker patterns (e.g., WebWorkers[1]), which however require a share-nothing architecture to preserve the semantics of the event-loop. Whenever multiple processes need to share state (e.g., to implement a stateful Web service), the data needs to be stored into a database or in an external repository providing the necessary concurrency control.

Concerning the second limitation, event-based programming requires the developer to deeply understand the event-loop runtime architecture and to write services with non-blocking mechanisms so as to break down long-running operations into multiple processing steps. Unfortunately, such mechanisms usually involve the adoption of programming techniques (e.g., nested callbacks, closures) which increase the complexity of developing even simple services. Moreover, while non-blocking techniques help increasing the throughput of a service, they also increase the latency of the responses. As a consequence, services need to be developed by carefully balancing blocking and non-blocking operations.

As an example, consider Fig. 1. The two code snippets in the figure correspond to two different implementations of a simple Node.JS Web service for calculating the n-th Fibonacci sequence number. The code in Fig. 1 (a) implements the Fibonacci function using the recursive algorithm by Leonardo da Pisa, while the one in Fig. 1 (c) implements the same algorithm exploiting non-blocking programming (using a hybrid synchronous/asynchronous algorithm). If a request is issued for a Fibonacci number which is greater than a fixed threshold (over which the standard recursive algorithm is known to block the event-loop for too long), the result is calculated using the non-blocking algorithm (`fiboA`). Otherwise, the blocking recursive algorithm (`fiboS`) is used. The non-blocking implementation does not use the stack for the entire recursion. Instead, it generates a series of nested events (through the `nextTick` runtime function call), each one corresponding to a single recursive function invocation. This fragments the execution flow of each request, as control is returned to the event-loop which can accept other incoming requests.

A comparison of the performance of the two services is given in Fig. 1 (b). For each implementation, a mixed workload of "light" (20th Fibonacci number) and "heavy" (35th Fibonacci number) requests is executed; the workload amounts to 100 requests per second. For the machine used in the experiment, the threshold

[1] `http://dev.w3.org/html5/workers/`

```
function fiboS(n) {
  if(n<2) return n
  else return fiboS(n-1)+fiboS(n-2)
}

http.createServer(function(req,res){
  var n = req.query
  res.end('result: '+fiboS(n))
}).listen(8080)
```

(a) Blocking Version

Heavy(%)	Bk (msg/s)	NBk (msg/s)
0.00	100.0	100.0
0.05	24.4	94.2
0.10	9.8	88.9
0.15	5.1	83.6

(b) Performance Comparison

```
function fiboA(n,done) {
  if(n<2) done(n)
  else process.nextTick(function() {
     fiboA(n-1, function(num1) {
       process.nextTick(function() {
         if(n>threshold)
           fiboA(n-2,function(num2){
             done(num1+num2)
           })
         else done(num1+fiboS(n-2))
       })})})
}

http.createServer(function(req,res){
  var n = req.query
  if(n>threshold)
    fiboA(n, function(value) {
      res.end('result: '+value) })
  else res.end('result: '+fiboS(n))
}).listen(8080)
```

(c) Non-blocking Version

Fig. 1. Blocking (Bk) vs. Non-blocking (NBk) Fibonacci Web Service in Node.JS

has been set to 30. Experiments have been performed with different percentages of heavy requests (up to 15%). Results show a notable difference between the two implementations. The blocking implementation achieves only low throughput compared to the non-blocking one, even with a low percentage of "heavy" requests. The reason is the sequential event-loop architecture: calling the `fiboS` function with values higher than the threshold keeps the event-loop blocked, thus preventing it from processing other clients' requests. This aspect, coupled with the impossibility of sharing a global memory space among different processes, constitutes a significant limitation for the development of high-throughput Web services using Node.JS.

3 The Programming Model of Node.Scala

The programming model of Node.Scala is similar to the one of Node.JS, as it features an implicit parallel programming model based on asynchronous callback invocations for the Scala language. However, blocking methods can be invoked without blocking the service, and concurrent requests running on different threads can safely share state. The goal is to let developers write services using the same assumptions (single-process event-loop) made on the Node.JS platform, while automatically and safely carrying out the parallelization to fully exploit multicore machines. This has the effect of freeing the developer from dealing with the issues identified in the previous section, while keeping all the benefits of the asynchronous programming model with implicit parallelism, overlapping I/O- and CPU-bound operations, and lock-free synchronization.

An example of a Node.Scala Web service (Fig. 2) similar to the one computing the n-th Fibonacci sequence number (Section 2) makes use of the two distinguishing features of Node.Scala, i.e., global stateful objects and blocking

```
def fiboS(n: Int): Int = n match {
    case 0 | 1 => n
    case _ => fiboS(n-1) + fiboS(n-2)
}
val cache = new NsHashMap[Int,Int]()
val server = new NsHttpServer(8080)
server.start( connection => // 1st callback
{
    val n = connection.req.query("n").asInstanceOf[Int]
    if( cache.contains(n) )
        connection.res.end("result: " + cache.get(n) )
    else
        server.nextTick( => // 2nd callback
        {
            val result = fiboS( n )
            cache.put (n, result)
            connection.res.end("result: " + result )
        })
})
```

Fig. 2. Simple Stateful Web Service in Node.Scala

synchronous calls. The stateful object (`cache`, of type `NsHashMap`) is used as a cache to store the values of previously computed requests. To perform the computation, a simple blocking function call (`fiboS`) is used. The algorithm used is the Scala-equivalent version of the recursive implementation from Fig. 1 (a). The service makes also use of two callback functions. As in Node.JS, the first callback represents the main entry point for the service, that is, the callback function that will be triggered for every new client's connection. The callback is passed as an argument to the `start()` method (implemented in the `NsHttpServer` class). The second callback used in the example is the argument to the `nextTick` method, which registers the callback to perform the actual calculation and to update the cache.

Each callback is invoked by the Node.Scala runtime whenever the corresponding data is available. For instance, as a consequence of a client connection, an HTTP request, or a filesystem access, the runtime system *emits* an event, which is put into the event-queue (i.e., into the list of all pending events to be processed). The event will then be taken from the queue by one of the threads running the event-loop, which will invoke the corresponding callback function with the received data passed as an argument. In this way, when a new client request is received, the runtime calls the first user-defined callback function passing the `connection` object as argument. The object (created by the runtime) can be accessed by all other nested callbacks, and holds all the details of the incoming request (`connection.req`), as well as the runtime object for generating the answer (`connection.res`).

The service is stateful because the first callback uses an object with global scoping, `cache`, which is not local to a specific client request (to a specific callback), but is global and thus shared among all parallel threads running the event loop. Node.Scala enables services to safely share state through a specific library of common Scala data structures, which are used by the runtime system to automatically synchronize multiple concurrent callbacks accessing the same shared data structure. The details of the runtime mechanisms allowing such safe implicit parallel processing are described in Section 4.

The second callback calls a synchronous method. In common event-loop frameworks such a blocking call would result in a temporary interruption of the event-loop, as discussed in Section 2. The parallel runtime system of Node.Scala overcomes this limitation using its architecture based on parallel event-loops. Therefore, blocking synchronous calls do not have a negative impact on Node.Scala service performance as they would have in traditional frameworks. Consequently, programmers can focus on developing the service business logic without having to employ complex non-blocking programming techniques to achieve scalability.

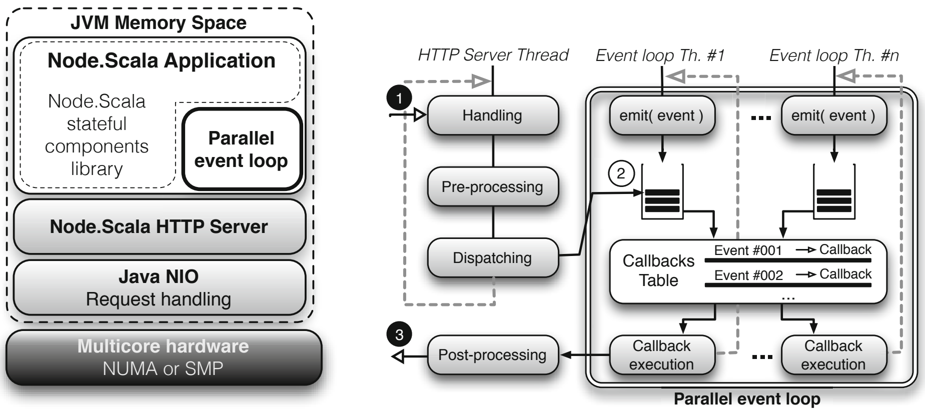
4 System Architecture

In this section we describe the system architecture (Fig. 3), focusing on the constructs that allow Node.Scala to safely parallelize request processing. Node.Scala uses a single JVM process with multiple threads to execute a Web service, granting shared memory access to the threads running the parallel event-loops. As illustrated in Fig. 3 (b), the request processing pipeline consists of tree stages: (1) handling, (2) processing, and (3) completion.

Request Handling. Incoming HTTP connections are handled by a dedicated server thread, which pre-processes the request header and emits an event to the parallel event-loop to notify a new request. All the operations performed by the HTTP server thread are implemented using the Java New I/O (NIO) API for asynchronous I/O data processing. Since each event-loop thread has a dedicated event-queue, the HTTP server thread adopts the *join-the-shortest-queue* policy to select which queue to fill.

Request Processing. Multiple event-loop threads concurrently process events generated by incoming requests. In particular, each event-loop thread removes an event from its local event queue, accesses the callback table associated with that event type, and executes the registered callbacks. New events generated by the execution of a callback are inserted into the local event queue of the processing thread. This mechanism ensures that all the events generated by a specific request are processed sequentially, according to the event-driven programming model. The callback table is automatically updated each time the execution flow encounters the declaration of a new callback function (see lines 7 and 13 in Fig. 2).

Request Completion. Responses are buffered using the `end` method. Once all events generated by a request are processed, the system replies to the client using the HTTP server thread, which also performs some post-processing tasks (e.g., generating the correct HTTP response headers and eventually closing the socket connection).



(a) High-level Architecture (b) The Parallel Event-loop

Fig. 3. Overview of Node.Scala

4.1 Thread Safety

Node.Scala Web services are automatically guaranteed to be thread safe. To this end, the runtime distinguishes between three types of requests: *stateful exclusive*, *stateful non-exclusive*, and *stateless*. This classification depends on the type of accesses to global variables[2]. If the processing of a request can trigger the execution of a callback that writes to at least one global variable, the request is considered stateful exclusive. Similarly, if the processing of a request can result in at least one read access to a global variable, the request is considered stateful non-exclusive. All other requests are considered stateless. As a consequence, a stateful exclusive request cannot be processed in parallel with other stateful requests. Instead, multiple stateful non-exclusive requests can be executed in parallel as long as no stateful exclusive requests are being processed. Finally, stateless requests can be executed in parallel with any other stateless and stateful request.

To perform this classification, Node.Scala intercepts class loading by means of the `java.lang.Instrument` API and performs load-time analysis of the bytecodes of each callback. Each user-defined callback is parsed by Node.Scala to track accesses to global variables. To speedup the analysis, methods of classes from the Node.Scala library are marked with two custom annotations: `@exclusive` and `@nonexclusive`.

Each time the analysis classifies a new callback as stateful exclusive or stateful non-exclusive, its bytecode is manipulated to inject all read (i.e., `ReadLock`) and write (i.e., `WriteLock`) locks necessary to ensure thread safety[3]. Lock acquisition instructions are injected at the beginning of the body of a callback, while lock

[2] Accesses to final values are not considered for the classification of requests.

[3] The semantics of `ReadLock` and `WriteLock` are defined in the documentation of the standard Java class library.

release operations are injected at the end. Therefore, the entire body is guarded by the necessary locks. This mechanism allows the event-loop thread to try to acquire all necessary locks at once. In case of failure, the event-loop thread can delay the execution of the callback and process events generated by different requests without breaking the programming model. After a predefined number of failed attempts, the event-loop thread blocks waiting for all the locks to avoid starvation. To prevent deadlocks, we associate a unique ID to each lock and we sort the order of the inserted lock acquisition and release instructions accordingly.

In the worst-case scenario (i.e., all callbacks always require the acquisition of the same set of exclusive locks) only a single event-loop thread can execute a single request at any given time. In this case, the performance of the service is comparable to the one of single-process, event-based frameworks that make use of sockets to communicate between different processes (e.g., Node.JS). In all the other cases, Node.Scala can effectively and safely parallelize the execution of callbacks, taking advantage of all available cores to increase throughput, as illustrated in the following section.

5 Performance Evaluation

To assess the performance of the Node.Scala runtime, we have implemented a Web service similar to the one presented in Fig. 2. Instead of the simple Fibonacci function, we used the entire set of CPU-bound benchmarks of the SciMark 2.0[4] suite, a well-known collection of scientific computing workloads. The service has been implemented using only blocking function calls, while both stateless and stateful services performance have been evaluated.

The machine hosting the service is a Dell PowerEdge M915 with four AMD Opteron 6282 SE 2.6 GHz CPUs and 128 GB RAM. Each CPU consists of 8 dual-thread modules, for a total of 32 modules and 64 hardware threads. Since threads on the same module share access to some functional units (e.g., early pipeline stages and the FPUs), the throughput of Node.Scala is expected to scale linearly until 32 event-loop threads. The system runs Ubuntu GNU/Linux 11.10 64-bit, kernel 3.0.0-15, and Oracle's JDK 1.7.0_2 Hotspot Server VM (64-bit).

The runtime performance of Node.Scala is measured using a separate machine, connected with a dedicated gigabit network connection. We use httperf-0.9.0[5] to generate high amounts of HTTP requests and compute statistics about throughput and latency of responses. For each experiment, we report average values of five tests with a minimum duration of one minute and a timeout of 5 seconds. Requests not processed within the timeout are dropped by the client and not considered for the computation of the throughput.

5.1 Stateless Services

To evaluate the performance of the Node.Scala runtime with stateless requests (i.e., with callbacks neither modifying nor accessing any global state), we have

[4] `http://math.nist.gov/scimark2/`
[5] `http://code.google.com/p/httperf/`

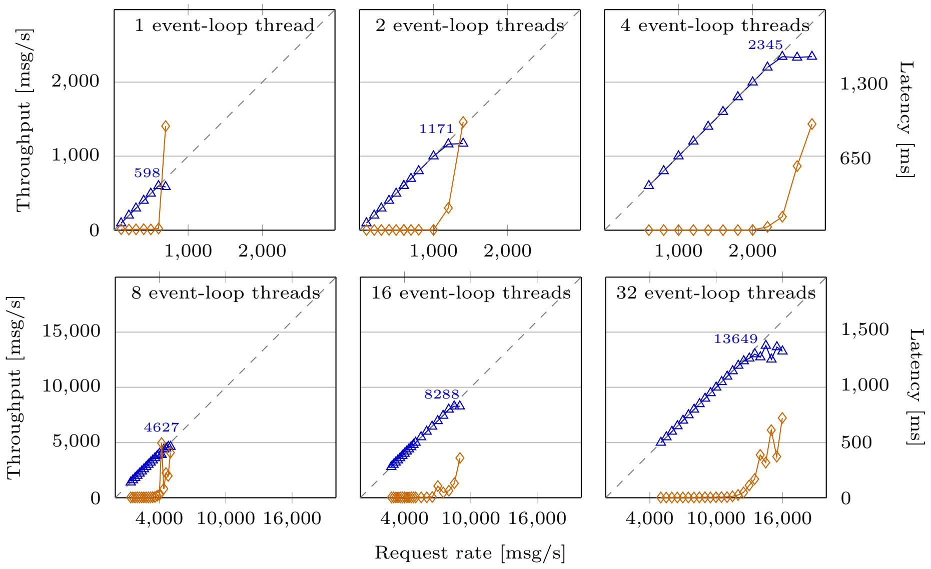


Fig. 4. Stateless service: throughput (—△—) and latency (—◇—) depending on the arrival rate and the number of event loop threads. The dashed reference line (- - -) indicates linear scalability.

disabled the caching mechanism in the evaluated service. Therefore, the service is a pure-functional implementation of the SciMark benchmark suite.

Fig. 4 illustrates the variation of throughput and latency of responses depending on the request rate and on the number of event-loop threads. The experiment with a single event-loop thread resembles the configuration of common single-threaded event-driven frameworks for Web services, such as Node.JS. In this case, the throughput matches the request rate until a value of 600 requests per second. During this interval, the latency remains below 10ms. Afterwards, the system saturates because the single event-loop thread cannot process more requests per unit time. As a consequence, the throughput curve flattens and the latency rapidly increases to more than one second.

Experiments with larger amounts of event-loop threads follow a similar behavior: the latency remains small as long as the system is not saturated, and it rapidly increases afterwards. The peak throughput measured at the saturation point scales almost linearly with the number of event-loop threads, up to a value of 13600 msg/s with 32 threads. This confirms the ability of Node.Scala to take advantage of all available CPU cores to improve the throughput of stateless Web services. Our experiments also confirm that the parallel runtime of Node.Scala allows the developer to use blocking function calls without any performance degradation.

5.2 Stateful Services

To evaluate the performance of stateful services, we enabled the caching mechanism of the Node.Scala service used for the evaluation, and we have tested it with

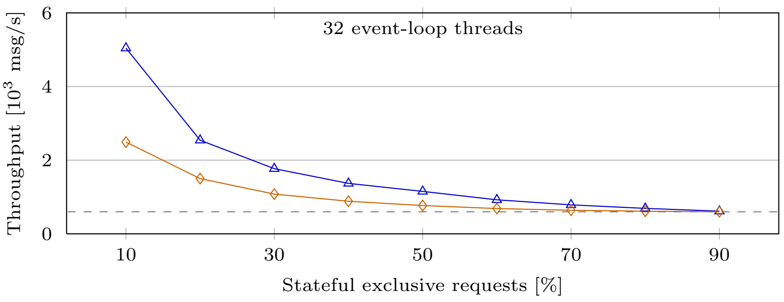


Fig. 5. Stateful services: throughput of SciMarkSf1 (—△—) and SciMarkSf2 (—◇—) depending on the percentage of stateful exclusive requests. The reference line (- - -) refers to the throughput achievable using a single event loop thread.

two different workloads. The first one (called `SciMarkSf1`) makes an extensive use of the caching mechanism, forcing the runtime to execute either exclusive or non-exclusive callbacks. The second one, (called `SciMarkSf2`) uses the caching mechanism only to store new data. Therefore, the second workload requires the runtime to process both exclusive and stateless callbacks.

The goal of both workloads is to assess the performance of the service in the worse possible cases, i.e., when the service is intensively using a single common shared object.

Fig. 5 reports the peak throughput of the two considered Web services, executed with 32 event-loop threads, depending on the amount of stateful exclusive requests. We do not report the values for corner cases, that is, 0% and 100%, because they are equivalent to the peak throughput presented in Fig. 4 for the cases with 32, respectively 1, event-loop threads. As reference, we plot a line corresponding to the performance with a single event-loop thread. When the number of stateful exclusive requests is high, performance is comparable to those of traditional, single-threaded, event-driven programming frameworks. However, when this number is smaller, Node.Scala can effectively take advantage of available cores to achieve better throughput.

6 Related Work

Web server architectures can be roughly classified into three categories [12]: thread-based, event-based, and hybrid [5,8]. The runtime of Node.Scala lies in the latter category, as it uses both event-loops and threads. A similar approach is represented by the SEDA architecture [16]. Both SEDA systems and the Node.Scala runtime feature multiple event queues and multiple threads. However, Node.Scala features a programming framework built on top of its runtime architecture which allows to develop stateful services, while SEDA's focus is only at the runtime level and does not handle state. There are several examples of event-based Web servers [11], as well as thread-based servers [15]. A long-running debate (e.g., [10,15]) comparing the merits of the two approaches has been summarized in [12]. In the same paper, an exhaustive evaluation shows that event-based servers yield higher throughput (in the order of 18%) compared to thread-based servers under certain circumstances. A previous attempt to parallelize event-based services has been presented in [17]. The approach proposed to manually annotate callbacks with color-based annotations, and then to schedule callbacks for parallel execution according to their color. In Node.Scala no manual intervention from the developer is needed to parallelize the service since callbacks do not have to be annotated. Akka[6] is a JVM framework for developing scalable Web services using the Actor model. Like Node.Scala, Akka supports HTTP and REST, as well as Java NIO. Differently, Node.Scala features a library to share state among different client requests, while Akka relies on Software Transactional Memory. Out of the realm of the JVM, event-based programming is implemented in several frameworks and languages. For instance, Ruby's EventMachine[7] allows services to be developed using the Reactor event-loop pattern [13].

7 Conclusion

In this paper we presented Node.Scala, a programming framework and a runtime system for the development of high-throughput Web services in Scala. Node.Scala features an event-based programming model with implicit parallelism and safe state management. Node.Scala developers have to deal neither with abstractions such as parallel processes or threads, nor with synchronization primitives such as locks and barriers. Instead, the developer can focus on the service business logic, while the Node.Scala runtime takes care of the parallel processing of concurrent requests. Services built with Node.Scala do not suffer from limitations of single-threaded event-based frameworks like long-running blocking methods and lack of support for shared memory. Thanks to the parallel event-loop architecture of Node.Scala, services leverage current shared-memory multicore machines with both stateless (i.e., purely functional) services and stateful ones. Stateless services exhibit controlled latency and linear scalability up to saturation. In stateful scenarios the parallel runtime system allows Node.Scala services to exploit a shared memory space and thus obtain better performance compared to other single-process solutions.

Our ongoing research focuses on extending the Node.Scala library with additional objects from the Scala standard library. To this end, we are experimenting with bytecode analysis techniques to automatically annotate Scala types with the `@exclusive`/`@nonexclusive` annotations used by the Node.Scala runtime to protect callback invocations. Finally, we are also consolidating the Node.Scala approach by generalizing its runtime system in order to port the Node.Scala parallel event-loop system to other JVM-based functional programming languages such as Groovy, Clojure, and Rhino JavaScript.

[6] http://akka.io/

[7] http://rubyeventmachine.com/

Acknowledgment. This work is partially funded by the Swiss National Science Foundation with the SOSOA project (SINERGIA grant nr. CRSI22_127386).

References

1. Bahi, J., Couturier, R., Laiymani, D., Mazouzi, K.: Java and Asynchronous Iterative Applications: Large Scale Experiments. In: Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–7 (2007)
2. Cardellini, V., Casalicchio, E., Colajanni, M., Yu, P.S.: The State of the Art in Locally Distributed Web-Server Systems. ACM Comput. Surv. 34, 263–311 (2002)
3. Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., Morris, R.: Event-Driven Programming for Robust Software. In: Proc. of the 10th ACM SIGOPS European Workshop (EW), pp. 186–189 (2002)
4. Fielding, R.T.: Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. thesis, UCI, Irvine (2000)
5. Haller, P., Vetta, A.: Actors That Unify Threads and Events. In: Murphy, A.L., Ryan, M. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 171–190. Springer, Heidelberg (2007)
6. Kinder, K.: Event-Driven Programming with Twisted and Python. Linux J. (2005)
7. Li, P., Wohlstadter, E.: Object-Relational Event Middleware for Web Applications. In: Proc. of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON), pp. 215–228 (2011)
8. Li, P., Zdancewic, S.: A Language-based Approach to Unifying Events and Threads. CIS Department University of Pennsylvania (April 2006)
9. Li, Z., Levy, D., Chen, S., Zic, J.: Auto-Tune Design and Evaluation on Staged Event-Driven Architecture. In: Proc. of the 1st Workshop on MOdel Driven Development for Middleware (MODDM), pp. 1–6 (2006)
10. Ousterhout, J.: Why Threads are a Bad Idea (for Most Purposes). In: USENIX Winter Technical Conference (1996)
11. Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: an Efficient and Portable Web Server. In: Proc. of the USENIX Annual Technical Conference (USENIX), p. 15 (1999)
12. Pariag, D., Brecht, T., Harji, A., Buhr, P., Shukla, A., Cheriton, D.R.: Comparing the Performance of Web Server Architectures. In: Proc. of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys), pp. 231–243 (2007)
13. Schmidt, D.C., Rohnert, H., Stal, M., Schultz, D.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, 2nd edn. Wiley (2000)
14. Tilkov, S., Vinoski, S.: Node.js: Using JavaScript to Build High-Performance Network Programs. IEEE Internet Computing 14(6), 80–83 (2010)
15. Von Behren, R., Condit, J., Brewer, E.: Why Events Are a Bad Idea (for High-Concurrency Servers). In: Proc. of the 9th Conference on Hot Topics in Operating Systems, vol. 9, p. 4 (2003)
16. Welsh, M., Culler, D., Brewer, E.: SEDA: an Architecture for Well-Conditioned, Scalable Internet Services. In: Proc. of the ACM Symposium on Operating Systems Principles (SOSP), pp. 230–243 (2001)
17. Zeldovich, N., Yip, E., Dabek, F., Morris, R.T., Mazires, D., Kaashoek, F.: Multiprocessor Support for Event-Driven Programs. In: Proc. of the USENIX Annual Technical Conference (USENIX), pp. 239–252 (2003)