Chapter 2

Background: single-threaded event loop

The event loop is a popular software design pattern for the development of so-called *reactive* applications, that is, applications that respond to external *events*. Examples of such applications are graphical user interfaces, and any class of services, including OS-level services and Web services. In this chapter we give an overview of event-based systems, with a particular focus on event-based server systems and their corresponding programming models in JavaScript.

2.1 Event-based concurrency

Event-based concurrency is a concurrency control model based on event emission and consumption. Historically opposed to the most common multi-threaded paradigm [53], event-based systems have gained notable attention in the context of service-oriented computing because of their simple but scalable design. In an event-based system, every concurrent interaction is modeled as an *event*, which is processed by a single thread of execution, called the *event loop*. The event loop has an associated *event queue*, which is used to schedule the execution of multiple pending events. Every event has an associated *event handler* function. Every event handler can access and modify a global memory space, which is shared with other event handlers. Being the event loop a single thread of execution, the event-based concurrency model presents the following properties:

- *Safe shared state.* Every event handler comes along with an associated state, which can be shared between multiple handlers. Since all the events are processed by the same thread of execution, the shared state is always guaranteed to be consistent, and data races are not possible (that is, concurrent modifications on the state from other threads never happen).

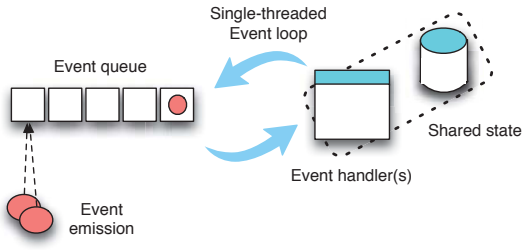

Figure 2.1. Event generation and execution in a single-threaded event loop.

- *Nonblocking processing.* Since all the events are processed by the same single thread of execution, each event handler must avoid long-running computations. A time-consuming event handler (e.g., a long CPU-intensive operation) would prevent other events from being processed, thus blocking the system. As a consequence of this design, blocking operations (for instance, I/O) are not supported by the runtime, and every operation must be asynchronous and nonblocking.

- *Ordered execution.* Events are added to the event queue with FIFO policy. This ensures that events produced with a given ordering are consumed (that is, executed) with the same ordering. Event ordering can be leveraged to chain successive events in order to split long-running computations, as well as to implement other forms of nonblocking computations[1].

The high-level architecture of an event-based system is depicted in Figure 2.1. Events are usually produced by external sources such as I/O operations (emitted by the operating system) or by the application interacting with the event loop (e.g., a GUI application controlling mouse movements). Every event handler is executed sequentially, and handlers have full control over the heap space of the process. Event handlers can invoke functions synchronously (by using the stack) or asynchronously, by emitting new events. This is depicted in Figure 2.2.

Event loop concurrency has some desirable characteristics for concurrent programming. Among others, the most important property of event-based systems is that data races are avoided *by design* allowing developers to make stronger assumptions on how each instruction is interleaved when concurrent requests are in the system. More formally, each event handler is guaranteed to run *atomically* with respect to other event handlers, and atomicity is naturally enforced by the single-threaded event loop.

Event-based concurrency also presents some drawbacks (with respect to multithreading). First, the single-threaded nature of the event loop system makes it hard to

[1]For instance, in GUI programming, the *mouseDown* event handler (generated when the user clicks on an icon) can be assumed to always run before the *mouseUp* event (generated when the button is released).

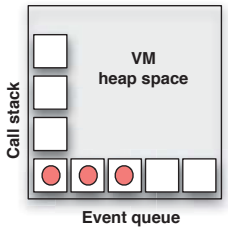

Figure 2.2. Event-based execution and shared memory in Node.JS and in the Browser.

scale event based systems on multicore machines without relaxing one of the properties above. In particular, the simplest way of scaling event-based systems on multicore machines is by replicating the event loop on multiple independent share-nothing processes. This has the drawback of *splitting* the memory space of event handlers between multiple processes, thus preventing event handlers from sharing a common memory space[2].

2.2 Event-based server architectures

Static HTTP servers are among the most common applications relying on single-threaded event-based concurrency. Services published on the Web need to guarantee high throughput and acceptable communication latency while facing fluctuating client workloads. To handle high peaks of concurrent client connections, several engineering and research efforts have focused on Web server design [49]. Of the proposed solutions, event-driven servers [61, 135] have proven to be very scalable, and their popularity has spread among different languages and platforms, thanks to the simple and efficient runtime architecture of the event loop [118, 115]. Servers of this class are based on the ability offered by modern operating systems to interact asynchronously (through mechanisms such as Linux's `epoll` [65]), and on the possibility to treat client requests as collections of events. Following the approach proper of event-based systems, in event-driven servers each I/O operation is considered an event with an associated handler. Successive events are enqueued for sequential processing in an I/O event queue, and processed by the infinite event loop. The event loop allows the server to process concurrent connections (often nondeterministically) by automatically partitioning the time slots assigned to the processing of each request, thus augment-

[2]Moreover, approaches based on process-level replication have other drawbacks with respect to memory footprint and service latency, as the communication between multiple processes can represent a source of performance degradation due to the need for exchanging -via memory copy- data structures (e.g., session data) between multiple processes.

ing the number of concurrent requests handled by the server through time-sharing. In this way, sequential request processing is overlapped with parallel 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). Being the server static, the lack of any integration with any dynamic language runtime makes the parallelization of such services trivial [135].

2.3 Event-based Frameworks and Web Programming

Of particular success in the domain of Web development are the so-called event-based frameworks, i.e., frameworks based on an event-based runtime systems for request handling and response processing. Such frameworks allow the developer to specify the service's semantics as a set of asynchronous callbacks to be executed upon specific events; a model which results particularly convenient when embedded in managed runtimes for languages such as JavaScript, Python, or Scala (popular examples of such frameworks are Node.JS [15], Akka [1], and Python Twisted [106]). Unlike other thread-based solutions, event-based frameworks have the advantage of not requiring the developer to deal with synchronization primitives such as locks or barriers, as a single thread (the event loop thread) is capable of handling a high number of concurrent connections. Furthermore, the asynchronous event-based programming style of such frameworks has influenced the programming model of client-side Web applications, as every Web application nowadays is using asynchronous event-based programming in JavaScript and HTML5 (i.e., AJAX [79]).

Event-based frameworks have recently become very popular, and several companies have started offering cloud-based PaaS hosting for event-based service development platforms (examples are Microsoft Azure [14] and Heroku [7]). However, despite their success, event-based frameworks still present limitations which could prevent them from becoming a mature solution for developing high-performance Web applications. In particular, the lack of structured parallelism forces the developer to implement complex ad-hoc solutions by hand. Such unstructured approaches often result in cumbersome solutions which are hard to maintain [126]. Some of the limitations of current programming models can be described as follows:

1. *event loop replication*: the event-based execution model offers only partial solutions to exploit parallel machines, since it relies on a single-threaded infinite loop. Such centralized architecture limits parallelism, since it is possible to exploit parallel machines only by replicating multiple processes using some Inter-Process Communication (IPC) mechanism for coordination. This approach prevents event-based applications from exploiting the shared memory present in multicore machines. Moreover, callbacks could have interdependencies, and cannot therefore be executed on multiple cores (indeed, the sole possibility to exploit multiple cores is by replicating the entire

event loop process). The common solution is to bind each specific connection to a specific core, preventing connections from exploiting per-request parallelism (i.e., to start multiple parallel processes to handle a single client request, using for instance more complex patterns such as, e.g., MapReduce [64] for generating the response).

2. *Shared state*: In single-threaded event loop concurrency event handlers can exploit a common shared memory space (that is, some data shared between multiple concurrent client requests). However, this becomes impossible when running multiple parallel event loop processes. The common solution to this issue is to use an external service to manage the state (e.g., Memcached [12]). However, such solution presents other limitations when the shared state is immediately needed by the application's logic. For instance, it is not always efficient to use an in-memory database to handle the temporary result of a parallel computation before serving the computation's result to the client.

Another consideration more specific to server-side event-based systems can be made considering the nature of server-side architectures. In particular, one of the main peculiarities of server-side parallelism over "traditional" parallel programming relies on the actual source of parallelism. In fact, in server-side development there is a *natural* source of parallelism represented by the abundant presence of multiple concurrent client requests which should be handled in parallel. Client requests could have interdependencies (for instance in stateful services), but in the great majority of cases each client request does not require the service business logic to use explicit parallelism (for instance using threads). This implicitly means that the logic of each request handler could be written using a plain sequential style, or an event-based style, but no explicit parallel programming abstractions need to be used for generating the client response while ensuring scalability.

We can call this peculiar characteristic of server-side development *natural parallelism* of Web services. Current existing frameworks do not fully exploit this source of parallelism, and instead require the developer to either manually manage parallelism (for instance by replicating processes or starting threads) or to rely on existing limited solutions usually offered by the cloud provider (for instance, several cloud providers allow users to specify the number of parallel replicas of a service that should be started automatically, at the cost of limiting the class of applications that can be potentially hosted in the cloud [109]).

2.4 Event-based programming models

The peculiarities of event-driven systems have promoted several programming models relying on event loop concurrency. Of particular interest in the scope of this Dissertation are all frameworks and libraries which allow Web services to be scripted and/or embedded within other language runtimes. Examples of programming models based or relying on the event loop include libraries and frameworks (e.g., Vert.X [26], Python

Twisted [106] or Java NIO [35]), as well as VM-level integrations such as Node.JS. Existing approaches can be divided into two main categories, namely programming models that explicitly require the developer to produce and consume events, and models that abstract the event loop by means of other higher-level concepts. In this section we give a brief overview of the two approaches, and we argue in favor of high-level ones.

2.4.1 Explicit event emission

Explicit loop manipulation programming requires the developer to explicitly produce and consume events in the form of callback functions. This can be done by explicitly adding a function to the global event queue, using a primitive called `async`[3]:

```
// Main event emission primitive for the single-threaded loop
async( Function to be added to the event queue , Optional arguments );
```

The primitive is nonblocking, and returns immediately after having added a function to the event queue. The queue is thread-safe, as other functions may be concurrently added by the underlying operating system. The primitive is the main mechanism used by event-based systems to introduce asynchronous execution, but does not provide any mechanism for relating functions with events. This is usually achieved with two higher-level core primitives, namely `on` and `emit`:

```
// Core event emission primitive
emit( Event label , Event arguments );

// Core callback registration primitive
on( Event label , Callback );
```

The `on` primitive can be used to specify one or more event functions (i.e., callbacks) to be associated with a specific event. In this Dissertation we adopt the convention that events are named using strings (called *event labels*), but other ways of specifying event identifiers also exist. We also adopt the convention that event emission and handling can be bound with existing object instances, and therefore expressions like `obj.emit/obj.on` mean that an event is being emitted and consumed only if the given object instance has registered a callback for the given event label. This is conceptually equivalent to having a callback table private to each object instance, but other design decisions can also be adopted (for instance, a globally shared callback table).

The `emit` primitive is used to notify the runtime that an event has happened, and its callback can be added to the event queue. The primitive is internally implemented using `async`, as depicted in Figure 2.3.

Events can also be associated with arguments. This is also presented in Figure 2.3, where at line 10 an event emitter is used to produce and consume events. The example corresponds to how incoming connections are handled by an event-based socket

[3] In other runtimes the `async` primitive can be called differently, e.g., `nextTick` in Node.JS

```
1  Object.prototype.emit = function(label, args) {
2    for(var callback in this.callbacks[label]) {
3      async(this.callbacks[label][callback], args);
4    }
5  }
6  Object.prototype.on = function(label, callback) {
7    this.callbacks[label].push(callback);
8  }
9
10 var obj = {};
11
12 obj.on('connection', function(fd) {
13   // open the file descriptor 33 and handle the request
14 });
15
16 // somewhere in the connection handling code.
17 obj.emit('connection', 33);
```

Figure 2.3. Usage and implementation of the `emit` primitive.

server like Node.JS. An object responsible for accepting the incoming request (`obj`) is registered to listen for an incoming connection using `on`. When the runtime receives a new connection on a listening socket the `'connection'` event is emitted, and the socket file descriptor on which the connection has been accepted is passed to the event handler. The callback associated with the `'connection'` event is then invoked with the actual value of the `fd` variable as argument (i.e., `33` in the example). Multiple calls to `emit` will result in multiple invocations of the callback function handling the event.

Event ordering ensures that multiple events emitted in a specific order will be executed in the same order they are produced. As a consequence, the following code depicted in Figure 2.4 will always result in the same console output.

The event emission primitives are low-level basic building blocks that can be used to build higher-level abstractions. Other primitives to wait for a specific event to happen, to deregister event handlers, as well as to trigger multiple events upon certain conditions (for instance using a pub/sub pattern [38]) exist. In the next Section we will discuss how such low-level primitives can be used as simple blocks for building other more convenient high-level abstractions.

2.4.2 Implicit loop manipulation and Node.JS

Although many libraries for explicit loop manipulation exist, explicit loop programming is usually considered a complex programming model, specific for low-level system development[4]. To overcome the complexity of the explicit interaction with the event loop, other high-level models have been proposed.

[4]The name of one of the most popular event-based frameworks, i.e., Python Twisted [106] is a perfect indicator for the complexity that certain event-based applications tend to have.

```
1  obj.on('data', function(data) {
2    console.log('the data is: ' + data);
3  });
4  obj.emit('data', 1);
5  obj.emit('data', 2);
6  obj.emit('data', 3);
7
8  // expected console output:
9  // the data is: 1
10 // the data is: 2
11 // the data is: 3
```

Figure 2.4. Ordered event emission using `emit`.

One of the most popular of such frameworks is Node.JS. Node.JS is a programming framework for the development of Web services in which the event loop is hidden behind a simpler programming abstraction, which allows the developer to treat event-driven programming as a set of callback function invocations, taking advantage of JavaScript's anonymous functions. 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. Consider the following *hello world* web service written in Node.JS JavaScript:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

The example corresponds to the source code of a very minimal yet scalable web service. No explicit loop manipulation is done by the user, who just has to provide a callback (in the form of an anonymous function, at line 2) to be executed for each incoming request. The Node.JS runtime is internally relying on explicit loop manipulation, as presented in Figure 2.5. The code in the example corresponds to the source code for the HTTP server of Node.JS, which is written in JavaScript itself. The code in this example registers multiple event handlers to handle the multiple phases of an incoming request handling. Since incoming requests may be sliced by the OS runtime into multiple *data chunks* (line 9), an event handler is registered to accumulate multiple chunks. Once the incoming data has been fully received, an explicit event emission is used (i.e., `newRequest`) to eventually invoke the user-provided callback.

Event-based programming coupled with the high performance of the V8 Engine makes of Node.JS one of the Web frameworks with the highest scalability in terms of concurrent connections handling [111]. Its programming model based on asynchronous callbacks fits well with the event loop as every I/O-event corresponds to a

```
http.prototype.createServer = function(callback) {
  // Create a new server instance
  var server = new runtime.HttpServer();
  // Event handler for any incoming connection
  server.on('connection', function(socket) {
    // create a buffer to store incoming data
    var buffer = runtime.createInputBuffer();
    // register an handler to accumulate incoming data
    socket.on('data', function(data) {
      if(data != '\n\r') {
        // accumulate data chunks
        buffer.push(data);
      } else {
        // once the data has been received, notify
        server.emit('newRequest', buffer);
      }
    });
  });
  // register an event handler to invoke the callback
  server.on('newRequest', function(data) {
      var request = runtime.createRequestObject(data);
      var response = runtime.createRssponseObject(data);
      // call the user-provided callback
      callback(request, response);
  });
}
```

Figure 2.5. Implementation of an event-based HTTP server

function callback invocation. As with event-driven systems, concurrent client requests are processed in parallel, overlapping I/O-bound operations with the execution of callbacks in the event loop. The event loop is implemented in a single process, while all the I/O-bound operations are carried out by the OS, thus, the developer has only to specify the sequential code to be executed for each event within each callback. Despite of the power of the event loop and the simplified concurrency management of the programming model, frameworks like Node.JS still present some limitations preventing them to exploit modern multicore machines.

2.5 Limitations of single-threaded event loop systems

When it comes to parallel execution, event-driven frameworks like Node.JS may be limited by their runtime system in at least two distinct aspects, namely (1) the impossibility of sharing a common memory space among processes, and (2) the difficulty of building high throughput services that need to use blocking function 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., WebWork-

```
1  var SIZE = 100000;
2  // Size of the subset to be returned
3  var SUBSET = 10000;
4  // Init the shared read-only array
5  var A = new Array();
6  for(var i=0; i<SIZE; i++) {
7    A.push(Math.floor(Math.random()*10000));
8  }
9  // Create the service and start it
10 var http = require('http');
11 http.createServer(function (req, res) {
12   // Get a subset of the array
13   var x = new Array();
14   for(var i=0;i<SUBSET;i++) {
15     var id = Math.floor(Math.random()*SIZE);
16     x.push(A[id]);
17   }
18   // Sort it
19   x.sort();
20   // Return the sorted array
21   res.end('sorted:' + x);
22 }).listen(1337, '127.0.0.1');
```

SUBSET	Throughput (msg/s)	Latency (ms)
10^2	2865.4	3.4
10^3	1799.3	6.9
10^4	253.8	40.3
10^5	23.3	421.2

Service performance for different values of SUBSET

Figure 2.6. Array sorting microbenchmark in Node.JS.

ers [8]), 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 in-memory 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.

As an example, consider the Web service in Figure 2.6. The code in the example corresponds to a simple stateful service holding an in-memory data structure (the array A, at line 5) which is "shared" between multiple concurrent clients. Each time a new client request arrives the service simply extracts a subset of the array and copies it to a new per-request object (the array x, at line 13). Once the subset array has been copied, the new array is sorted by using the `sort` JavaScript builtin function, and the resulting sorted array is sent back to the client. The service corresponds to a simple yet significant microbenchmark showing how the two limitations introduced above may affect the Node.JS model:

- Long-running computations: sorting the array corresponds to a small yet significant operation that can significantly affect the performance of the service. This is shown in the table presenting the performance of the service for different values of the SUBSET variable (which corresponds to the size of the subset of the array to be copied)[5]. As the table clearly shows, an even small size of the array is enough

[5]The microbenchmark is executed on two distinct machines using a recent version of Node.JS

to affect the performance of the service in a very relevant way. In particular, the throughput drops and many client requests fail because of a timeout. Requests that do not timeout are affected as well, as the average latency of the service also grows.

- Shared state: to reduce the slowdown introduced by the sorting operation, the only solution for current event-based systems would be to offload the sorting operation to another event loop running in another process. Not having the ability to safely share state among processes, however, the state would need to be either replicated (that is, cloned into multiple processes), or to be sent to other processes as needed (e.g., by using an external database). Both approaches may result in a waste of space (that is, increased memory footprint) and might affect the service's latency. Moreover, from a programming model point of view the event-based system will lose part of its appeal because of the introduction of message passing and explicit coordination between processes.

In the next sections of the Dissertation we will discuss how the PEV model can be used to solve or attenuate the two issues without changing the single-threaded programming model.

(v0.10.28) with the `ab` HTTP client using with the following configuration: `ab -r -n 10000 -c 10 http://127.0.0.1:1337/`.