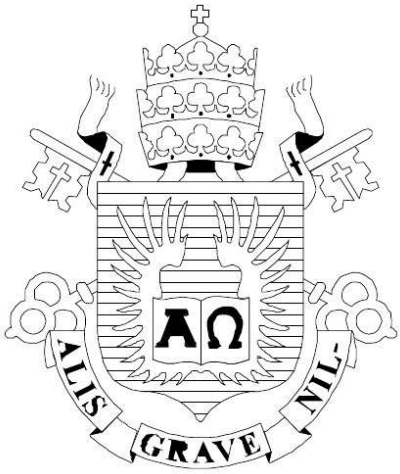

PUC

ISSN 0103-9741

Monografias em Ciência da Computação

nº 01/2021

Events in the MPA system

Renan Santos

Noemi Rodriguez

Roberto Ierusalimschy

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 01/2021 ISSN: 0103-9741
Editor: Editor: Prof. Carlos José Pereira de Lucena Abril, 2021

Events in the MPA system

Renan Santos, Noemi Rodriguez and Roberto Ierusalimschy

rmiranda@inf.puc-rio.br, noemi@inf.puc-rio.br, roberto@inf.puc-rio.br

Abstract. Implementing the event-driven paradigm is most commonly done using event loops. In this study, we give an overview of the concepts surrounding the paradigm and discuss the problems of having multiple event loops and the difficulties presented by blocking operations in an event based system. We then present the MPA system, a software used for industrial automation, and debate the usage of events in its applications.

Keywords: Events; event loop; MPA.

Resumo. A implementação por trás da programação orientada a eventos é mais comumente feita usando laços de eventos. Nesse estudo, nós apresentamos uma descrição dos conceitos por trás desse paradigma e discutimos sobre os problemas encontrados na interação entre múltiplos laços de eventos e sobre as dificuldades inerentes a operações bloqueantes em sistemas baseados em eventos. Nós também apresentamos o MPA, um software usado na área de automação industrial, e debatemos o uso de eventos em suas aplicações.

Palavras-chave: Eventos; laço de eventos; MPA.

In charge of publications:
PUC-Rio Departamento de Informática - Publicações
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: publicar@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

In this paper, we discuss the event-driven programming paradigm. We provide a basic description of its concepts and establish a common vocabulary to address them. It is important to firmly ground each concept and its denominations, since most of the terms of the event-driven paradigm are heavily overloaded. In our analysis, we focus on the implementation details regarding event loops and the interactions between distinct loops. Mainly, we discuss the difficulties of having two concurrent event loops running in the same program and the problems of dealing with blocking operations inside single-threaded event-driven programs.

In the second part of this paper, we analyze the MPA system (a software used for industrial automation) as a case study to discuss the event-driven paradigm. We examine the inner workings of the system in view of its two subsystems, the User Interface and the Engine, both containing task schedulers that behave similarly to event loops. We then provide suggestions as to what can be changed in the MPA system in order for it to accommodate the event-driven programming paradigm. We make these suggestions targeting specifically the Engine implementation and the visual language used to create its automation programs.

2 Events

In some applications, the flow of execution is heavily dependent on external stimulus. For example, in graphical user interfaces, programs must react to button presses and other user actions. Likewise, in soft real-time systems, the behavior of a program is determined by the occurrence of events that trigger the execution of callbacks. Instead of adopting the standard procedural programming style, these applications typically follow the event-driven paradigm. In this model, an event is an external occurrence such as a sensor reading, a received message, or an user action. Moreover, events must be bound to handlers, that is, callbacks that get executed once an event occurs. Due to this structure, the event-driven paradigm is associated with inversion of control, as program execution does not follow a sequential list of instructions determined by the programmer. Instead, execution is controlled by a framework or runtime environment that reacts to events by calling handlers.

For some types of programs, such as GUIs, using events facilitates the process of developing clearer programs, since there is a natural correspondence between user input and resulting behavior with events and handlers. Also, for systems with soft real-time or non-blocking I/O requirements, the event-driven paradigm can be an alternative to thread-based models and the concurrency problems associated with it. Program architectures based on events are easier to scale and manage than ones that rely on directly creating new threads. Additionally, in small scale programs, events can be a lightweight replacement for the complexities of multithreading. Finally, events can also be used to wrap blocking or time-consuming operations into non-blocking ones. Later in this section, we will provide a more in-depth discussion on some of these usages.

2.1 Nomenclature

Before we continue, we will deviate a little from the main topic of this paper to address the nomenclature confusion surrounding the word *reactive*. We use *reactive* to describe the general flow of execution in event-driven programs, but the term has been used alongside at least four other contexts in the programming world. Let's analyze them.

- Functional reactive programming [3, 1] is a paradigm that focuses on the propagation of change through continuous time-varying values (called *behaviors*). For instance, when we define $x = y + z$ in a functional reactive program, if the values of y or z ever change, then x will be updated to receive the new computed sum of y and z. FRP programs also feature discrete occurrences in time such as button presses and mouse clicks, called *events*. The dependency relations between behaviors, events, and data guides much of the flow of execution in functional reactive programs and defines the inversion of control inherent to the model. For this reason, FRP is also called event-driven, but with higher level abstractions.

- Reactive programming libraries, such as ReactiveX [6], provide extensions over the Observer pattern in order for programs to monitor and react to values that change discretely over time. These tools allow imperative languages to behave in an event-driven manner, while also featuring common functional traits such as *map* and *filter* functions.

- Reactive systems [5] is a denomination for systems that are responsive (provide quick and consistent response times), resilient (responsive in face of failure), elastic (adapted to varying workloads), and message driven. The term describes the desired characteristics of software for the current needs of the industry, without imposing on how these programs should be developed.

- Real-time computing (RTC) defines software that must abide to real-time constraints. RTC languages such as Esterel [2] and Céu [7] follow the synchronous reactive programming paradigm. In these languages, the notion of time is abstracted into logical ticks, with computations between ticks being assumed instantaneous. Albeit heavy and restrictive, this assumption allows synchronous reactive programs to appear deterministic in face of concurrent events and, therefore, be easier to reason about. Moreover, SRP languages commonly provide statements that allow programs to wait on a given condition, synchronize concurrent executions, send/receive messages, etc.

As we have seen, *reactive* has become an overloaded term, and the same is true for *event-driven*. Many tools, libraries, and languages have been called event-driven. Since this is such a general denomination, we could certainly analyze these models in terms of events and handlers. However, we must not confuse the higher level abstractions with the core of the concept. In essence, event-driven programs operate in terms of events and handlers, and that is it. This core model is the subject of our research in this paper, and we will consider these characteristics when referring to the event-driven paradigm from now on.

2.2 Implementation

Most systems that support events rely at their core on an event loop. An event loop continuously checks for the occurrence of events and arranges for the execution of bound handlers. Event loops are low-level constructions and, for this reason, can sometimes be hidden from the programmer. For example, in some GUI programming libraries, the programmer does not need to explicitly manipulate the loop. The library will take care of starting, stopping or closing the loop when required, and the user needs only to code handlers and register them for events. Some other libraries, however, explicitly expose the loop and allow programmers to handle it freely. Moreover, event loops can be single or multithreaded. A single-threaded loop checks for events, queues the appropriate handlers, runs queued handlers, and repeats the process. Multithreaded loops can run multiple handlers concurrently and/or use multiple threads to continuously check for events internally.

That being said, how are events implemented in event loops? In the Unix world, where "everything is a file descriptor", events almost always represent some kind of I/O, and we deal with them using a `select`-like operation[1]. A `select` call monitors the status of multiple file descriptors at once, waiting until one or more of them become "ready" for some class of I/O operation [8]. It receives as parameters a set of file descriptors and a timeout indicator, and blocks until one or more of the file descriptors can be used or the timeout elapses. If the timeout is set to zero, the function checks the file descriptors and returns immediately. In a standard single-threaded event loop, all events get processed until the system blocks in a `select` operation waiting for external events. In multithreaded versions, one thread can be in charge of handling the `select` while others run the handlers and the main loop. These implementations work well, but lead to some issues when dealing with interactions between multiple event loops. In the following subsections, we discuss these problems.

2.2.1 The problem with multiple event loops

There is no easy way to merge calls to `select` from two distinct codebases without altering the codebases internally. For instance, imagine you have an event-oriented program with a main loop that uses a `select`-like operation to perform non-blocking I/O on files. When a read or write is complete, the loop guarantees the proper handlers will be queued to run. Sometime during development, you are forced to use a third-party library to deal with TCP connections. This library also uses `select` and an event loop to provide non-blocking operations. How do you integrate your original loop with the one from the library? The fact is, depending on how these loops are implemented, you won't be able to merge them without making some compromises.

The problem we described only exists because `selects` are hidden inside the implementation of event loops and cannot be directly manipulated by programmers. This characteristic is essential to the event-driven model since it guarantees inversion of control and allows non-blocking operations to behave as expected. If the programmer were himself in charge of the call to `select`, he would be able to arbitrarily insert events in the list of

[1]There are multiple incarnations of `select`-like operations across operating systems: *epoll* for Linux, *kqueue* for BSD, *IOCP* for Windows, etc. For simplicity, when referring to `select`-like or `select` in this paper, we are referring to this class of operation. Moreover, we assume that a `select` operation is unique in each OS, or at least that all file descriptors operate interchangeably with them. For example, Linux has the *epoll* and *poll* functions, buth both can operate on the same file descriptors.

events to be observed. In this case, we wouldn't be able to assume the programmer would not (incorrectly) block execution on I/O.

Given the situation where we have multiple `selects` and cannot alter the loop implementation, there are some suboptimal solutions to the merging problem. First, we can resort to *polling*. Event loops usually have a `step` function that runs the loop body once without blocking (zero timeout on `select`). In order to integrate multiple loops, we could iterate through each of them calling `step` and then sleep for a short time before repeating the process. However, this is the same as busy waiting. If the sleep timeout is too short, we burden the CPU. If it is too long, the program ends up being not very responsive. Alternatively, both loops could run in separate threads and signal their events to a main thread. This solution is good for performance, but introduces the problems of dealing with threads and shared memory (e.g. avoiding data races and synchronizing execution flows). We can avoid memory sharing issues by "faking" message passing. For example, one thread could write into a pipe to signal an event while the main `select` watches over that pipe's file descriptor. It is clear, however, that this would diminish performance. Moreover, in both solutions threads would become a program requirement, which could prove a problem to strictly single-threaded loops.

If we had some control over the APIs of the loops, we could try to solve the multiple-`selects` problem in another manner. With loops providing ways for programmers to manipulate (list, add, remove) the set of file descriptors of their `select` calls, we could redirect the work of one loop to another and, therefore, run only one loop. However, we would also have to register the internal handler for each added file descriptor, because, for example, a loop that only deals with TCP connections may not know how to behave when receiving file descriptors for pipes. Depending on the language and the implementation of the libraries, engineering a solution to this problem could prove to be a large programming project in itself.

2.2.2 Event loops and blocking operations

What if you already have an event loop and need to integrate it with blocking or time consuming operations? How do you prevent these operations from jamming single-threaded loops? The straightforward solution is to use a thread pool [4]. Blocking operations can be called on worker threads that signal the event loop with the results when finished. Then, the results get passed as arguments to the appropriate handlers. This method allows blocking operations to behave as non-blocking. Besides, the fact that they are running in separate threads is merely an implementation detail unbeknownst to the programmer at higher abstraction levels.

The `libuv` C library uses this mechanism to deal with I/O operations on files. Since the library is available to multiple operating systems, its developers chose not to use straight non-blocking operations for files due to the incongruencies between the available APIs. Using threads was the alternative.

We showcase some of `libuv`'s capabilities in the example in Listing 1, where we read a file and print its contents. First, as seen in line 9, `libuv` explicitly requests the programmer to start (run) the main loop. It also allows the user to stop and resume the execution of the loop. To open the file, `libuv` provides the `uv_fs_open` function, that, as other file system functions in the library, receives a callback to be executed when the operations finishes. The `onOpen` function (line 14) simply calls the `uv_fs_read` function passing

```
1  uv_buf_t* buf;
2  uv_fs_t openReq, readReq;
3  uv_loop_t* loop;
4
5  int main() {
6    <initialize buffer>
7    loop = uv_default_loop();
8    uv_fs_open(loop, &openReq, "path", O_RDONLY, 0, onOpen);
9    uv_run(loop, UV_RUN_DEFAULT);
10   <cleanup>
11   return 0;
12 }
13
14 void onOpen(uv_fs_t* req) {
15   assert(req->result >= 0);
16   uv_fs_read(loop, &readReq, req->result, buf, 1, -1, onRead);
17 }
18
19 void onRead(uv_fs_t* req) {
20   if (req->result == 0) {
21     uv_fs_t closeReq;
22     uv_fs_close(loop, &closeReq, openReq.result, NULL);
23   } else if (req->result > 0) {
24     printf("%s", buf.base);
25   } else {
26     <error>
27   }
28 }
```

Listing 1: Using `libuv`

the `onRead` callback. The `onRead` function can either print what was read (line 24) or close the file after EOF (lines 20 to 22). Note that, when calling `uv_fs_close`, `NULL` was passed as an argument instead of a callback. In such cases, `libuv` handles the function call synchronously. Execution of the program flows as follows: the loop starts and blocks waiting for the file to open; when the file is opened, `onOpen` is called; then, the loop blocks waiting for the file to be read; when the file is read, `onRead` is called, first to print what was read, then to close the file; finally, the loop ends and `uv_run` returns because there are no other registered events.

3 MPA

MPA is an industrial automation system used to develop and run applications that implement process control algorithms. It uses visual programming languages such as Sequential Function Charts (SFC – IEC 61131) and spreadsheet-like tables to provide programming and documenting tools to engineers. It is also used for reference and monitoring purposes by plant operators and other professionals responsible for supervising processes. In its essence, an MPA application is a collection of user-implemented diagrams and plant parametrization data. We will use the MPA system as a basis to discuss the event-driven paradigm.

3.1 Event loops in the MPA implementation

We can divide the MPA system in two subsystems: the User Interface (often called the IDE) and the Execution Engine. The User Inteface allows users to develop, compile and manage their flowchart-based applications, also acting as a control panel during execution. The Execution Engine is responsible for effectively loading and running the compiled application code. It connects to and operates actionable systems, such as pumps and valves, according to the logic of the application's diagrams. Regarding the flow of execution, the Interface and the Engine run in different processes, and can even run in different machines. For this reason, they communicate with each other using CORBA. The Engine reports to the Interface the current state of the diagrams' execution, and the Interface can signal the Engine to pause, resume, or stop execution while allowing the user to monitor the state of any given diagram. Essentially, the User Interface is a passive subsystem that reacts to commands from the user and the Execution Engine.

The Interface and the Engine use third-party libraries containing hidden event loops that cannot be manipulated by the programmer. Both use the OiL[2] library as their CORBA object request broker. The OiL loop uses the *cosocket*[3] library to monitor sockets over a `select`-like operation. The User Interface, additionally, uses the IUP[4] library when building its GUI (graphical user interface). Let's analyze how these multiple loops interact.

In the Interface, the IUP loop is in control of the flow of execution. It periodically calls the `step` function of its OiL instance to manage remote CORBA calls. Basically, the Interface uses the polling technique we described previously to integrate its two event loops. Because the MPA system has soft real-time requirements, a modest level of delay when reaching external systems is not a problem; thus, the polling technique mentioned in 2.2.1 can be used without burdening the CPU or degrading the application. Nevertheless, the developers were restricted to this method of solving the "two event loops" problem because they did not have access to system threads or the inner workings of the IUP loop.

Additionally, the Execution Engine has its own scheduler of cooperative threads (coroutines) to manage diagram execution. A flowchart diagram, along with other sub-entities, can be mapped to different threads in the scheduler. Each time one of these threads perform I/O, the scheduler adds the thread to a "pending I/O" queue. When all threads get blocked in this manner, the scheduler opens its connection with the industrial plant and performs (synchronously) the required I/O operations in a single batch[5]. Once that is done, the scheduler wakes the "pending I/O" threads.

The scheduler in the Engine only yields control to its OiL loop (and, therefore, only receives messages from the User Interface) when a *sleep* operation from the flowchart is executed. It recovers control as soon as the time of the *sleep* operations expires. As currently implemented, the diagram scheduler has priority over the OiL loop. In fact, if the current diagram does not sleep, the User Interface will never be updated with the state of the execution. Moreover, the Interface's monitoring facilities (pause, resume, etc.) will not work since the Engine will never receive these commands. The MPA team is currently exploring the usage of event-based programming in the visual language as an alternative

[2] `http://webserver2.tecgraf.puc-rio.br/~maia/oil`

[3] `https://github.com/renatomaia/cothread`

[4] `http://webserver2.tecgraf.puc-rio.br/iup`

[5] Performing these operations in a single batch is important because accessing the industrial plant is costly. Better to do all at once than constantly request to access the industrial plant.

to requiring *sleep* operations and loops in all diagrams. We discuss these (among other) options in the following subsection.

3.2 Suggestions

One of the main issues with the current implementation of the MPA system is diagram complexity. Large applications demand diagrams to contain several loops and sleep operations combined, which can often become a problem for maintainability and the general understanding of the flow of execution. In essence, diagrams become too coupled and less cohesive as size increases. Our goal is to ensure low coupling and high cohesion, no matter diagram size. For that, we suggest implementing event-based constructs in the diagram's visual language and in the MPA Engine.

Aiming to decrease complexity levels in diagrams configures as a quality-of-life improvement for those who have to develop and maintain MPA applications. Perhaps this reason alone would not merit the development effort required to implement events. However, as we explained in earlier sections, there is also the issue of loops being required to contain sleep operations, or else there is no communication with the plant and the User Interface. This is a more serious problem, since it can lead to apparently correct diagrams that do not perform as expected. As we will further explain, implementing events can solve both the sleep requirement and diagram complexity issues, which is why we recommend it.

Diagrams in the MPA system use loops and sleeps to repeat monitoring patterns. For example, imagine an application that checks the temperature in a container every five seconds and opens an emergency valve if the container gets too hot. If we had event-driven programming, we could easily model the *check temperature* scenario as an event, with a corresponding (small) diagram as its handler. Also, we could use the Interface to program events to be triggered once every x seconds. In this case, we would be effectively taking advantage of and reusing the loop behind the implementation of the event-driven paradigm, while also eliminating sleeps by creating a *repeater* construct for the system. This improves the complexity of diagrams by dividing MPA applications into multiple small diagrams, instead of a single central one. Naturally, it is easier to manage multiple small diagrams than one monolithic one. Additionally, to eliminate sleep requirements, we could use the nature of event loops to add fixed plant and Interface checks at each x loop steps. It is easy to parameterize such operations, as they would behave like repeated handlers. These changes would force the application to always yield to check on incoming messages from the plant or the Interface, without having to rely on users.

As to the implementation of these changes in the MPA system, we must address some specific points separately:

- First, diagrams must be parameterizable. For example, for the *check temperature* event, a container instance must be provided as an argument to the event. In turn, the diagram will receive this container as a parameter. This is a basic property of the event-driven paradigm.

- The essential building block (literally) to be added to the visual language is the *trigger* block. As the name implies, it triggers an event and may receive values as arguments to be passed to the corresponding handler. This block allows a big diagram to be compressed into a small one that controls trigger stages for multiple events.

- The *trigger* block must trigger events asynchronously and events must be preregistered to handlers in the Interface. When an event is triggered, the event loop internally queues its diagram for execution.

- Repeaters set handlers to be triggered once every x seconds. We can choose between providing a repeater native to the inner working of the event loop, or implementing it using a loop with a sleep and a trigger. Either way, it must be hidden from the user.

- Despite the addition of repeaters, the sleep block can be left mostly unchanged. A sleep basically yields back to the loop. The loops resumes once at least x seconds have passed. (The "at least" part is important, since there is no way to guarantee exact time, as other handlers can take an arbitrary number of seconds to execute.)

- As to the Lua implementation, some changes would need to be done to the current scheduler that uses the *cothreads* library. Mainly, we would need to modify its event loop to work with priorities in order to schedule diagrams and internal events separately. The interaction with the plant and the Interface would remain poll based, as we explained earlier. We cannot define plant events because communication with the plant is costly. If the plant itself could send monitoring messages, then we could add plant events to the event loop.

It is important to say we don't *need* to use events to deal with the sleep requirement, instead we can insert "artificial sleeps" in the diagram. We could accomplish that by running a path analysis algorithm in the diagram. For every path with no explicit sleep, we would need to insert an implicit sleep. This guarantees interaction with the plant and Interface always occurs, regardless of user error. Essentially, this is a solution that does not alter the MPA architecture.

Finally, we note that early in our experiments with the MPA system, we tried to implement a new diagram block to perform the trigger action. We soon realized this would not be a trivial task, mainly due to the project's complexity and to our insufficient understanding of the code's architecture. For this reason, we changed our approach to the research and chose to study the implementation of the system and interview its developers and maintainers. With the knowledge gathered from those meetings and analysis, we were able to describe and contextualize the functioning of the MPA system as presented in the previous section. We then started to think about the problems and demands associated with the MPA system and whether or not the event-driven paradigm would be recommended to address them. As it turned out, events would be a great fit. In the previous paragraphs, we presented the results of our research and our suggestions for future development. Implementing these suggestions is a large programming endeavor that requires rewriting and refactoring a great part of the project's code. On top of the event-related changes, some of this process may involve developing new methods of communication between the Engine and the User Interface (e.g. REST based APIs and message queuing protocols), while also updating the usage of older programming techniques (e.g. using callbacks instead of software components when dealing with streams of information). That being said, we believe this study will serve as reference for future work with events in the MPA system.

References

[1] Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., and Meuter, W. d. A survey on reactive programming. *ACM Comput. Surv.* (2013).

[2] Boussinot, F., and de Simone, R. The esterel language. *Proceedings of the IEEE* (1991).

[3] Elliott, C., and Hudak, P. Functional reactive animation. *SIGPLAN Not.* (1997).

[4] Lea, D. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* 1999.

[5] Reactive Manifesto Contributors. Reactive manifesto, 2020. [Online; accessed 06-September-2020].

[6] ReactiveX Contributors. ReactiveX, 2020. [Online; accessed 06-September-2020].

[7] Sant' Anna, F., Ierusalimschy, R., and Rodriguez, N. Structured synchronous reactive programming with céu. In *Proceedings of the 14th International Conference on Modularity* (2015), Association for Computing Machinery.

[8] Stevens, W. R. *UNIX Network Programming: Networking APIs: Sockets and XTI.* 1997.