Threading by Appointment

Christoph M. Kirsch

Department of Computer Science
University of Salzburg, Austria
ck@cs.uni-salzburg.at

Abstract. We propose a concurrent programming model called threading by appointment (TAP). Unlike traditional threads, TAP threads can only communicate with system components or other threads by appointment. For example, a TAP thread cannot simply try to access some shared resource. Instead, a TAP thread must make an appointment with the shared resource in advance. Only at the time of the appointment, the thread can actually access the shared resource. The desired start time and duration of an appointment may either be provided by the thread or computed by the TAP runtime system. The duration of an appointment must be finite. Instantaneous appointments are possible but only allow a single atomic operation per appointment. TAP threads cannot deadlock because the duration of appointments is finite. Race conditions may occur for making appointments but can be avoided by making appointments independently of the system's progress. Threading by appointment makes the process of determining the time instants for system interaction explicit and may therefore help to increase the determinism of concurrent applications as well as the efficient use of resources. Interesting questions arise such as how to schedule appointments as opposed to how to multiplex resources such as the CPU, and how to define structured programming elements that support threading by appointment.

1 Introduction

This paper gives an overview of some of the existing models for thread-based and event-driven concurrent programming, mostly in the domain of Internet applications, and informally presents some new ideas on how to overcome some of the limitations. However, the presented ideas have not been verified in practice and thus require more work, in particular, experimental studies. Nevertheless, we feel that the presented ideas point out potentially interesting research directions that are worth mentioning and investigating.

Threading libraries such as POSIX, Linux, or Java threads are widely used in practice but suffer from a number of problems such as non-deterministic behavior, presence of race conditions, and potential deadlock. We propose a concurrent programming model called *threading by appointment* (TAP) that supports deterministic I/O behavior and eliminates deadlock and some forms of race conditions. TAP threads can only communicate with system components or other threads by appointment. In order to access a shared resource such as an I/O device or shared memory, a TAP thread must make an appointment with that resource in advance. Only at the time of the appointment, the thread can access the resource even if the resource is available earlier. The duration of appointments must be finite, which eliminates the potential for deadlock. The notion of time for appointments can be anything from real time to, e.g., some form of CPU time. For example, in the latter case time may advance as soon as all threads are blocked [1]. A TAP thread *blocks* when the thread yields the CPU to show up for an appointment. Instantaneous appointments are also possible but only allow a single atomic operation per appointment. Multiple instantaneous appointments can be made at the same time instant. Race conditions may occur for making appointments but can be avoided by making appointments independently of the system's progress.

The time of an appointment may either be suggested by the thread or computed by the TAP runtime system, which maintains so-called *TAP calendars* for the appointments. Only the request by a thread for a non-instantaneous appointment may be turned down if the requested time has already been assigned to some other thread. In this case, the thread and the runtime system have to negotiate an alternative time. TAP calendars not only enable *fair* and *coordinated* access to shared resources and thus potentially reduce thrashing but also support resource-aware scheduling strategies [2]. TAP calendars can be seen as a generalization of event queues used in event-driven concurrency models such as SEDA [3]. TAP threads therefore offer a thread-based programming model that has the potential for utilizing the efficiency of an event-driven implementation while reducing the chances of thrashing.

We study event-driven and thread-based concurrency models using a simple process model called the *reactor-scheduler-process* (RSP) model, which has originally been used in the context of real-time programs [4]. Event-driven and thread-based models have previously been compared and shown to be essentially duals of each other [5] in terms of how events are handled. With the RSP model, we focus on a different aspect, i.e., the fact that both models essentially handle and create the same type of events. We try to provide an alternative perspective on the problem of whether events [6] or threads [7] are better for a given application. In fact, we argue that a key problem is not only how to handle events but also what kind of events to handle and when. TAP on RSP actually reveals that threading by appointment may just as well be introduced as *events by appointment* or, even more abstract, as *processing by appointment*. However, in order to keep the discussion sufficiently concrete we have chosen not to do so in this paper. In Section 2, we introduce the RSP model and present work related to threading by appointment in Section 3 using the RSP model as foundation. In Section 4, we introduce threading by appointment and discuss a simple version of so-called cooperative TAP threads. Section 5 gives an outlook on potential research directions.

2 The RSP Model

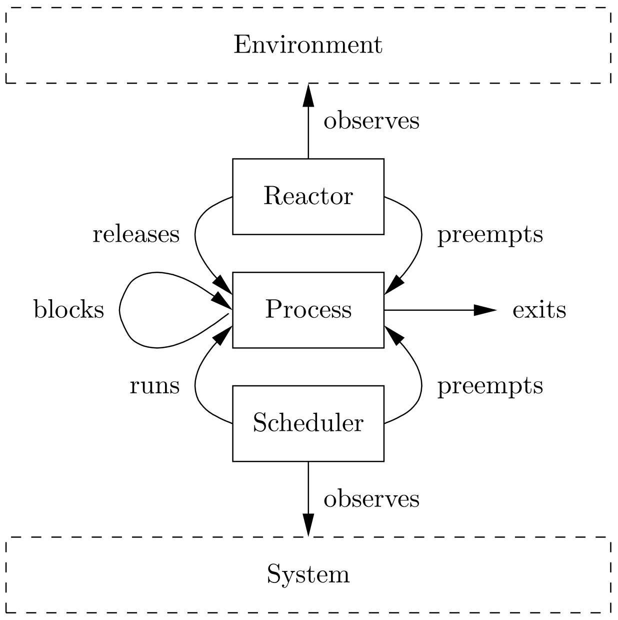

Fig. 1. The RSP model

The purpose of the RSP model is to understand the flow of control in the scheduling core of an operating system (i.e., RSP models task management in the sense of [8] but not stack management). RSP abstracts from any state-related aspects such as memory management and protection. Figure 1 depicts the flow of control in an RSP system. The *reactor* observes the environment of the system for occurrences of events such as system clock ticks or the arrival of new data. Upon the occurrence of an environment event, the reactor may release blocked processes that have previously expressed an interest in the event through some system call. The *environment* of an RSP system is the physical world around the RSP system that is visible through the system's timing and I/O devices. We will later extend the reactor to observe the RSP system itself, i.e., the scheduler and the processes. Most modern operating systems including many real-time operating systems implicitly do that, which explains why the flow of control in such systems is so hard to understand. The environment is an important parameter of an RSP system and needs to be defined carefully.

The reactor may release any number of processes but does not actually run processes. The function of the reactor is to relate environment events and processes, not processes and resources. This is up to the scheduler in an RSP system.

The *scheduler* observes the RSP system, i.e., the reactor and the released processes, and selects released processes to run. More precisely, in analogy to the reactor, the scheduler observes the RSP system for the occurrences of so-called *system events* such as system clock ticks (time-slicing scheduler) or the release or blocking of a process. Upon the occurrence of a system event, the scheduler may reconsider which processes to run. Note that a system clock tick is the only type of event that is not only an environment event but also a system event because system clock ticks are not under the control of the system but only refer to real time (unless the system has some means to modify the system clock's speed). If the reactor and the scheduler need to run at the same time instant, then the reactor has priority over the scheduler.

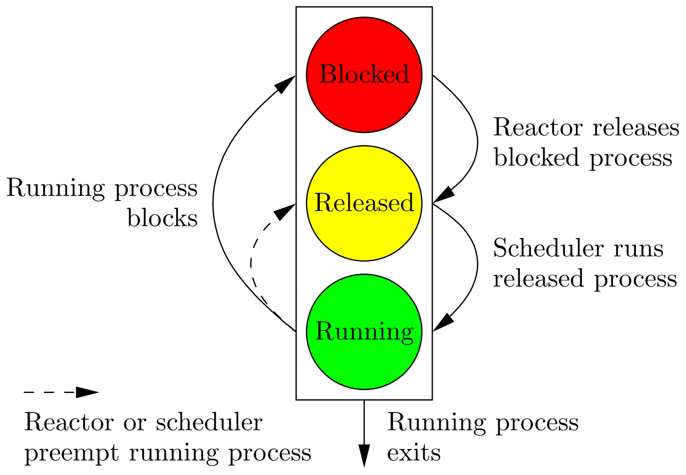
Blocked
Released
Running
Reactor releases blocked process
Scheduler runs released process
Running process blocks
Reactor or scheduler preempt running process
Running process exits

Fig. 2. The traffic light of process states

The scheduler may run any released process or decide to idle but cannot make a process block or exit. This is entirely up to the process, which may block by invoking a system call. Importantly, blocking in the RSP model not only refers to blocking for I/O or some semaphore as in most operating systems but merely refers to the transfer of control back to the reactor. Blocking and exiting is cooperative. Only a process can block itself but not the reactor or the scheduler. Figure 2 depicts the process states of the RSP model as a traffic light. Each state transition is under control of a unique system component except process preemption, which can be initiated by the reactor or the scheduler. Since process blocking and exiting is cooperative, the RSP model does not support termination and signals. Although this restriction could easily be removed, we have chosen not to do so because of simplicity and because termination and signaling are orthogonal to the issues discussed here.

The RSP model helps to understand in which way system components depend on each other and what kind of information flows between them. For example, changing the reactor may not only change the overall system behavior but also the requirements on the scheduler. In recent work such as [3] and [2], we observe that improving a system's performance (e.g., throughput) essentially requires improving the quality and quantity of information that flows between reactor, scheduler, and processes. However, these improvements occur at the cost of increased system complexity in the sense that the dependency between reactor, scheduler, and processes is increased. Consequently, system behavior is determined by an increasing number of factors, which further complicates system composition, analysis, and maintenance.

3 Related Work

We further illustrate the RSP system model by discussing how existing concurrent programming models relate to the RSP model. In traditional preemptive threading models such as POSIX, Linux, or Java threads, the flow of control can be quite complex. To simplify the discussion we do not consider signals and other control-based mechanisms that modify the behavior of a thread from outside the thread such as putting the thread to sleep or terminating the thread. Given some multi-threaded program, suppose that a thread A that is currently running attempts to acquire a lock on some shared resource. We model A by an RSP process that blocks at the attempt (even in the uncontended case). Note that blocking is a logical concept in the RSP model rather than an implementation technique. The time instant when the RSP process A blocks depends on many factors such as CPU performance and load and, in particular, on the process implementation, which is not under the control of the reactor or scheduler. As soon as the process is blocked, the reactor takes over and decides how to proceed. We can only model this behavior with a reactor that can observe system events in addition to environment events because a process that blocks constitutes a system event. If the lock is not available, the reactor marks A to wait for the lock and then relinquishes control by invoking the scheduler. If the lock is available, the reactor marks the lock as held by A and then releases A before relinquishing control to the scheduler. Now, suppose that some time later a thread B that actually holds the lock releases the lock. In RSP terminology, B is another RSP process, which will block when B attempts to release the lock. As soon as the process blocks, the reactor is invoked. If there is no process waiting to acquire the lock, the reactor marks the lock as available and releases B. Otherwise, if there is a process waiting to acquire the lock such as A, the reactor will choose one such process and release it in addition to B. To be fair this has to be done in FIFO order, i.e., the first process that was blocked to wait for the lock will be the first process to be released when the lock becomes available. Surprisingly, many older thread libraries and even kernels actually release *all* waiting threads causing the infamous "thundering-herd problem" if there are many threads waiting for the lock. In this case, the burden is put onto the scheduler to "catch" the herd and put it back to sleep again except for one of those threads.

Handling I/O requests in the RSP model is done in a similar way as acquiring locks. For example, a thread that attempts to read from some shared resource such as an I/O device is modeled as an RSP process that blocks at the attempt before invoking the reactor. If the I/O device is available, the reactor releases the process and invokes the scheduler. Otherwise, the reactor marks the process as waiting for the I/O device and invokes the scheduler. Later, when the I/O device becomes available, the reactor is invoked and releases the process. Interestingly, the original request for I/O is a system event but is implicitly turned into an environment event if the I/O device was not available at the time when the thread requested it. In summary, the traditional threading model requires a reactor that observes environment and system events. In particular, threads can control when the reactor releases processes by acquiring and releasing locks. As a consequence, there is a cyclic control-flow dependency between the reactor, the scheduler, and the processes.

Event-driven concurrency models are essentially equivalent to thread-based models in terms of the RSP model. They only differ in the way they manage events and memory, which is not exposed in the RSP model. Given some event-driven application, suppose that a callback function A will be invoked when an environment event r occurs. The event indicates that some data has arrived that needs to be read and processed by A. We model this relationship by a reactor that maintains an event queue and a hash table that associates events with callbacks (i.e., event handlers). For example, the event handler table associates r with A. The callback A is modeled by an RSP process. When r occurs, the reactor appends r to the event queue. If the system is idle, the reactor removes the first event from the queue, say, r, releases A because r is associated with A in the event handler table, and finally invokes the scheduler. Traditional event-driven models are non-preemptive, i.e., there is at most one callback function executing at the same time. Thus the scheduler trivially chooses and runs A immediately. If an environment event occurs the reactor preempts A to append the event to the event queue but does not release any other processes. Therefore, the scheduler can trivially resume the execution of A after the reactor is finished. When A completes, A blocks and the reactor is invoked. The completion of A constitutes a system event that the reactor appends to the event queue if there is a callback associated with the event. Therefore, similar to thread-based models, the reactor needs to observe system events in order to capture event-driven models. Finally, the reactor removes the next event from the event queue, if there is any, and releases the associated callback's process. If the event queue is empty, the system idles with the reactor observing environment events. Now, suppose that the completion event of A is in fact associated with some callback function B in the event handler table. As a consequence, the reactor eventually removes the completion event of A from the event queue and releases B. In this way, chains of callbacks can be constructed that can perform more complex tasks, which may involve multiple I/O interactions such as reading an http request and writing back some html page as an answer.

	Thread-Based Model	Event-Driven Model
Reactor	thread queue, locks	event queue, state machine
Scheduler	thread queue, priority-driven	-
Process	thread	callback

Table 1. Traditional instances of the RSP model

Table 1 summarizes the relationship of the traditional thread-based and event-driven models in terms of the RSP model. Note that variations of thread-based and event-driven models exist, e.g., middleware event services may support preemptive rather than cooperative callbacks using a possibly priority-driven scheduler.

	Capriccio [2]	SEDA [3]
Reactor	event queue, boolean or spin locks	stage, event queue, state machine
Scheduler	thread queue, resource-aware	thread pool, resource-aware
Process	thread	thread

Table 2. Refined instances of the RSP model

In order to maintain the automatic stack management [8] of the thread-based model while utilizing the efficiency of an event-driven implementation, thread-based models can be implemented on top of an event-driven model by adding an appropriate scheduling mechanism. The state thread library [1] implements such a model using a cooperative scheduler. The Capriccio thread library [2] goes even further by using a resource-aware scheduler and more advanced stack management. The other direction is also possible: for example, the responsiveness of an event-driven model can be improved by utilizing a pool of preemptive threads as alternative to callbacks, e.g., in the SEDA model [3]. In addition, SEDA uses multiple stages of event-processing machinery that can be modeled by multiple RSPs. Table 2 relates Capriccio and SEDA in terms of the RSP model, suggesting that recent implementations of thread-based and event-driven models become more and more alike. There are also hybrids of thread-based and event-driven models that have successfully been implemented, e.g., [9] and [8].

Table 3 relates two real-time programming models in terms of the RSP model. xGiotto [10] is an event-driven programming language with so-called logical execution times for tasks. In xGiotto, a task is the basic unit of functionality. A task computes from its input and state some output and a new state. An xGiotto

	xGiotto Model [10]	Synchronous Reactive Model [11]
Reactor	event filter and queue, virtual machine	event queue, state automaton
Scheduler	task set, real-time	-
Process	E code and task	synchronous reactive program

Table 3. Real-time instances of the RSP model

program defines the task functionality as well as when a task is invoked (i.e., for which kind of events) and when a task is terminated (i.e., when the output of the task will be available, independently of the task's execution time). A typical analysis of an xGiotto program involves a so-called time-safety check in which, for a given system configuration, all tasks are checked to complete execution before their output is made available. xGiotto introduces the notion of event scoping, which allows the programmer to encode an environment assumption on the arrival behavior of events in the actual structure of xGiotto programs. xGiotto can be seen as a language that offers a restricted form of threading by appointment using real time as the notion of time for appointments. The runtime system of xGiotto implements an RSP reactor that uses an event filter and queue for event scoping and a virtual machine, which is an extended version of the Embedded Machine [12], that executes the timing code generated by the xGiotto compiler. The RSP scheduler in a xGiotto runtime system is a real-time scheduler that handles the xGiotto tasks released by the virtual machine. An xGiotto task and its compiler-generated timing code, which is so-called E code executed by the virtual machine, make up a process in the RSP terminology.

The synchronous reactive model [11] uses the notion of zero time for real-time programming. In this model, computation takes zero time. For any event that occurs, a synchronous reactive program computes a reaction in zero time. In practice, such a program must be checked to complete any reaction before another event occurs. The RSP reactor in a runtime system for synchronous reactive programs is typically a compiler-generated state automaton that uses an event queue and non-preemptively dispatches code to compute the reactions. In this case, the RSP scheduler is obsolete. The synchronous reactive model produces deterministic and highly efficient code. However, similar to the event-driven model in the non-real-time world, system composition and distribution, and thus code maintenance and reuse are difficult. Nevertheless, synchronous reactive programming may in fact be interesting to the non-real-time event community.

4 The TAP Threading Model

Threading by appointment (TAP) is a concurrent programming model that supports deterministic I/O behavior, eliminates deadlock, and reduces the number of possible race conditions. TAP threads can only communicate with system components or other threads by appointment. In order to access a shared resource such as an I/O device or shared memory, a TAP thread must make an appointment with that resource in advance. Only at the time of the appointment, the thread can access the resource even if the resource is available earlier. The duration of appointments must be finite, which eliminates the potential for deadlock. The notion of time for appointments can be anything from real time to, e.g., some form of CPU time. Instantaneous appointments are also possible but only allow a single atomic operation per appointment. Multiple instantaneous appointments can be made to occur in some order at the same time instant. Race conditions may occur for making appointments and thus for getting access to shared resources. However, there may be strategies for making appointments that avoid race conditions.

The time of an appointment may either be suggested by the thread or computed by the TAP runtime system, which maintains so-called *TAP calendars* for the appointments. Only the request by a thread for a non-instantaneous appointment may be turned down if the requested time has already been assigned to some other thread. In this case, the thread and the runtime system have to negotiate an alternative time. In the following, we propose, as a starting point, a simple form of TAP threads called *cooperative TAP threads* that require the runtime system to determine the times of appointments. The details on more complex forms of TAP threads are future work.

4.1 Cooperative TAP Threads

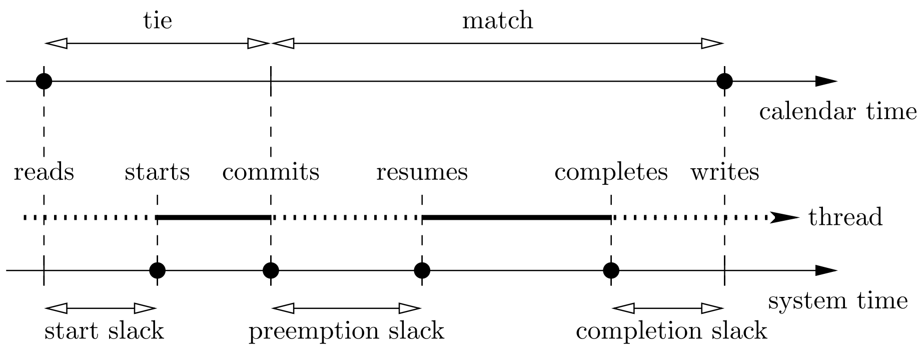

Fig. 3. A cooperative TAP thread

We introduce cooperative TAP threads in terms of the RSP model. Figure 3 shows the execution of a cooperative TAP thread A, which begins with a read operation on some I/O device. We model A by an RSP process. The instantaneous appointment for the read operation was obtained by A some time earlier. The read operation is performed by the reactor although the operation is part of the thread code. After the operation is finished the reactor releases A and invokes the scheduler, which may select other processes to execute first for the duration of the so-called *start slack*. Eventually, A is selected by the scheduler and executes for some time until A *commits* to a new instantaneous appointment for a write operation. The time of the appointment is determined by the so-called *secretary*, which is part of the reactor. The secretary maintains for each shared resource a so-called *calendar* of appointments. The time until A commits to an appointment is called *tie*. During the tie the scheduler has no information from the secretary when A will need access to some shared resource again. The time after A commits until the appointment is called *match*. During the match the scheduler can exploit the information about the appointment for A. When A has committed to an appointment, the scheduler will only resume the execution of A if there are no other appointments scheduled before the appointment for A. The time until the execution of A is resumed is called the *preemption slack*. Cooperative TAP threads can be preempted but only at the commit operation. Before and after the commit, cooperative TAP threads always execute non-preemptively. Thus cooperative TAP threads are safe to use shared memory before and after the commit and therefore do not need non-instantaneous appointments. However, accessing shared memory in this way avoids the appointment mechanism. Eventually, A *completes*, i.e., cooperatively gives up the CPU to the reactor and blocks. The rest of the time until the appointment is called *completion slack*. Depending on the notion of time used for appointments it may be possible that a thread does not complete before its next appointment, e.g., when using real time. For simplicity, we use a notion of time here that avoids such a situation altogether. Time advances if all released threads have committed to an appointment, and all released threads have completed that have committed to instantaneous appointments at the same time and there are no other appointments scheduled earlier. For example, suppose that there are four released threads, A, B, C, and D. Moreover, A and B both have instantaneous appointments at time 1 whereas C has an appointment at time 2 and D does not have an appointment yet but will commit to an appointment at time 3. Before time can advance all threads must have committed to an appointment, i.e., D runs and commits to an appointment at time 3. Then, time advances as soon as A and B have completed regardless of C. Consequently, cooperative TAP threads cannot miss appointments using this notion of time.

The appointments in the TAP model advance in what we call *calendar time* whereas any other thread activity such as starting, committing, resuming, and completing threads advances in *system time*. The fact that a TAP thread needs to make an appointment for I/O means that the time instant when the I/O occurs is somewhat decoupled from the time instants when the thread happens to run. Threading by appointment makes the process of coordinating the access of threads to shared resources explicit. The strategy of the secretary for making appointments determines the system's I/O behavior as well as the system's resource utilization. The more independent this strategy is from the system's load and performance, the more deterministic the system's I/O behavior will be. On the other hand, the more the secretary considers the system's load and performance in making appointments, the more efficient the system may be utilized. Finding and understanding strategies for the secretary is important future work. For cooperative TAP threads, appointments could be made in a way that sequences of appointments that result in particularly efficient resource utilization are remembered using some form of *calendar caching* and then repeated whenever possible. Another potentially interesting strategy to improve system utilization could be to reorder I/O calls using appointments but only up to the point where some given higher-level thread semantics is not violated, similar to the out-of-order execution of machine instructions.

	State Threads [1]	Cooperative TAP Threads
Reactor	event queue, state machine	calendar, secretary
Scheduler	thread queue, non-preemptive	thread set, calendar-driven
Process	thread	thread

Table 4. State threads and cooperative TAP threads in the RSP model

Cooperative TAP threads relate to state threads [1] as shown in Table 4. State threads implement a cooperative threading model on top of an event-driven architecture. With state threads, time advances in a similar way than described above. In RSP terminology, everytime all released state threads have completed, i.e., all RSP processes are blocked, the reactor releases all processes for which I/O is available. Then, the scheduler takes over and runs the released processes non-preemptively in some order. Each process performs the I/O, computes something, and eventually blocks on another I/O operation. When all processes have completed, the reactor is invoked again to release another round of processes for which I/O is available.

5 Outlook

We have presented a model for concurrent programming called threading by appointment and used a simple process model called RSP to study and relate the threading-by-appointment model to existing thread-based and event-driven concurrent programming models. The purpose of the threading-by-appointment model is to make the process of determining the time instants for system interaction explicit. There are many interesting open problems that arise in the context of threading by appointment. For example, (1) concepts for more general TAP threads such as preemptive TAP threads have not been investigated; (2) finding strategies for making appointments is important future work; (3) new structured programming elements that support threading by appointment may help to develop TAP-based systems; and (4) formal notions of TAP thread composition may have the potential to support a more compositional style of concurrent programming.

We have recently started working on the design and implementation of a threading library for cooperative TAP threads and a web server running on top of the library for benchmarking. Depending on the results, we plan to generalize cooperative TAP threads to some notion of preemptive TAP threads that also support non-instantaneous appointments. We will also study structured programming elements in order to support the development of correct-by-construction TAP threads.

References

1. Shekhtman, G., Abbott, M.: State threads for Internet applications. http://state-threads.sourceforge.net/ (2000)
2. von Behren, R., Condit, J., Zhou, F., Necula, G., Brewer, E.: Capriccio: Scalable threads for Internet services. In: Proc. SOSP. (2003)
3. Welsh, M., Culler, D., Brewer, E.: SEDA: An architecture for well-conditioned, scalable Internet services. In: Proc. SOSP. (2001)
4. Kirsch, C.: Principles of real-time programming. In: Proc. EMSOFT. Volume 2491 of LNCS., Springer (2002) 61–75
5. Lauer, H., Needham, R.: On the duality of operating system structures. Operating Systems Review **13**(2) (1979) 3–19
6. Ousterhout, J.: Why threads are a bad idea (for most purposes). In: Proc. USENIX. (1996)
7. von Behren, R., Condit, J., Brewer, E.: Why events are a bad idea (for high-concurrency servers). In: Proc. HotOS. (2003)
8. Adya, A., Howell, J., Theimer, M., Bolosky, W., Douceur, J.: Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In: Proc. USENIX. (2002)
9. Druschel, P., Pai, V., Zwaenepoel, W.: Flash: An efficient and portable web server. In: Proc. USENIX. (1999)
10. Ghosal, A., Henzinger, T., Kirsch, C., Sanvido, M.: Event-driven programming with logical execution times. In: Proc. HSCC. Volume 2993 of LNCS., Springer (2004) 357–371
11. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer (1993)
12. Henzinger, T., Kirsch, C.: The Embedded Machine: predictable, portable real-time code. In: Proc. PLDI. (2002) 315–326